

Documento de diseño de Software

Proyecto:

Digital Input Mini

PCB:

DigitalInput_Mini.PrjPCB

Versión:

V0.0

Índice

Introducción....4
Entradas y salidas físicas....5
Asignación de pines del microcontrolador....6
Máquina de estados....8
  Esquema de entradas y salidas....8
Jerarquía de drivers/útiles....9
Estructura de datos de entradas y salidas....10
  Estructura básica de un objeto IO....10
  Arrays de Input/outputs....11
    Inicialización de los arrays REG_IO y OnChange_IO....14
    Asignación de variables para las IRQ....14
    Funciones asignadas a los métodos:....15
  Super objeto con métodos....16
    Funciones de los métodos del superobjeto:....16
Gestión de la máquina de estados....17
  Módulo de estados:....17
    Tipo de datos estado:....17
    Función Info_a_Salidas....17
    Función ProcesarEstado....18
    Funciones auxiliares para comprobar eventos de entrada:....18
      Función Check_Encoder....18
      Función Check_Ch_ComStat....19
Útil del Encoder....20
  Tipos de datos....20
  Funciones asociadas a la clase:....21
    Rutina de servicio de interrupción....21
    Función Init:....22
    función READ:....23
Útil de Output 4 LEDs y MUX_Channel_Selector....23
  Tipos de datos....23
  Funciones asociadas a los métodos:....24
    Funciones INIT....24
    Funciones Write:....25
Útil y driver de EEPROM....25
  Driver:....25
    Tipos de datos....25
    Funciones asociadas a los métodos:....26
      Init....26
      Read....27
      Write....27

Constructor:....28
Util 28
Tipos de datos....28
Útil y driver de OLED Screeen Display....30
Driver....30
Tipos de datos....30
Función init:....31
Secuencia de inicialización de la pantalla OLED:....32
Función Write Command y Write Data....34
Constructor:....35
Util OLED:....36
Tipos de datos:....37
Tipos de datos: archivo de imágenes....38
Generación de los arrays de imagen:....39
Driver de GPIO....41
Tipos de datos....41
Funciones:....42
Constructor....42
COMUNICACIONES....44
Driver UART unificado....44
Estructura de la cola de trabajos:....46
Tipos de datos:....46
Trabajo de transmisión....46
Clase cola....47
Métodos de la cola:....48
Init: 48
Pop:....48
Push: 49
Arbitro de cola....49
Driver TXRX....50
Clase del driver:....50
Init:....51
Start y stop:....52
Read y write:....52
Util commando IN....52
Rutina de servicio de interrupción del DMA – RX....52
Código de la rutina de interrupción:....53
Reconocedor de comandos:....53
Util Transmitter....54
trasmitWrite....54
Rutina de servicio de interrupción del DMA - TX....56
Listado de comandos....57
Estructura del archivo:....57
Lista de comandos:....57

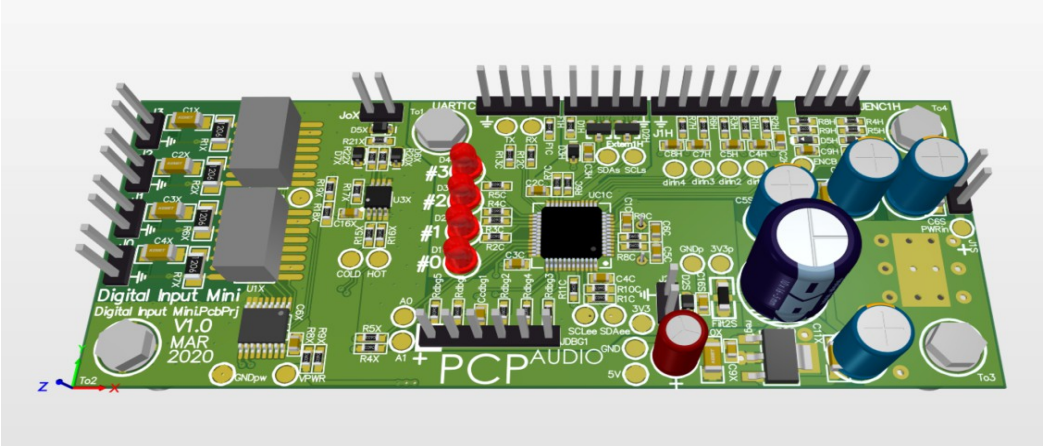
Versión "finalizable".....59
En esta versión "finalizable" se deberían hacer los siguientes cambios:.....59
Control de versiones.....60

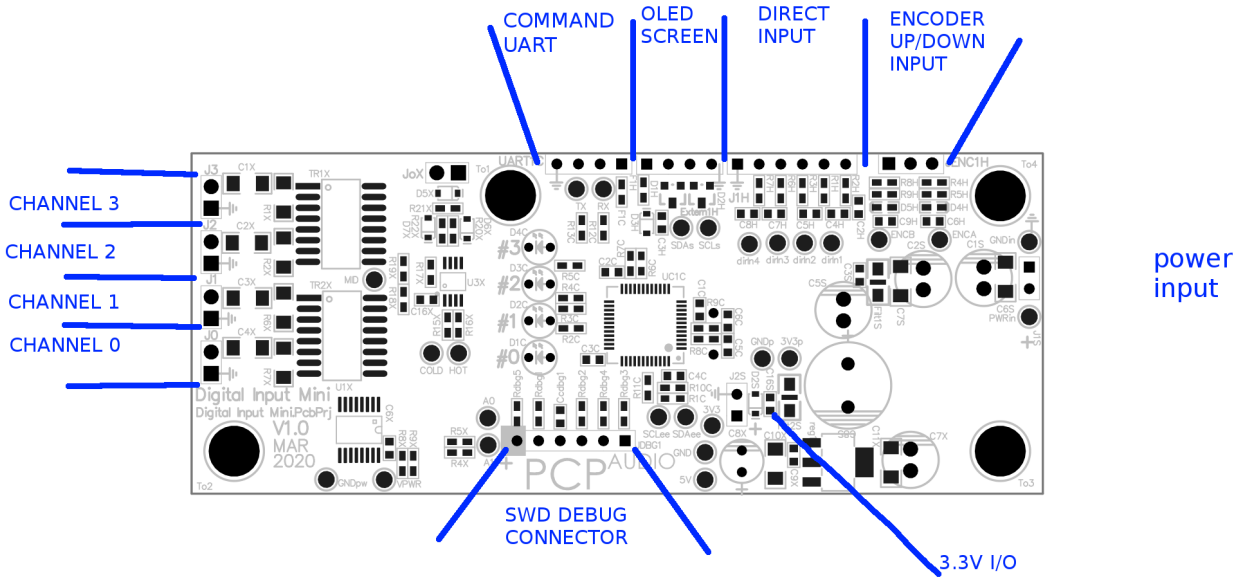
Introducción

Este documento tiene como finalidad recoger todos los aspectos trascendentes del diseño de software.
Se describe la máquina de estados, el esquema de entradas/salidas, los drivers, los útiles, y sus tipos de datos y una mínima descripción de las funciones más importantes.
Incluye qué opciones se han valorado, y porque unas has sido aceptadas y otras rechazadas.

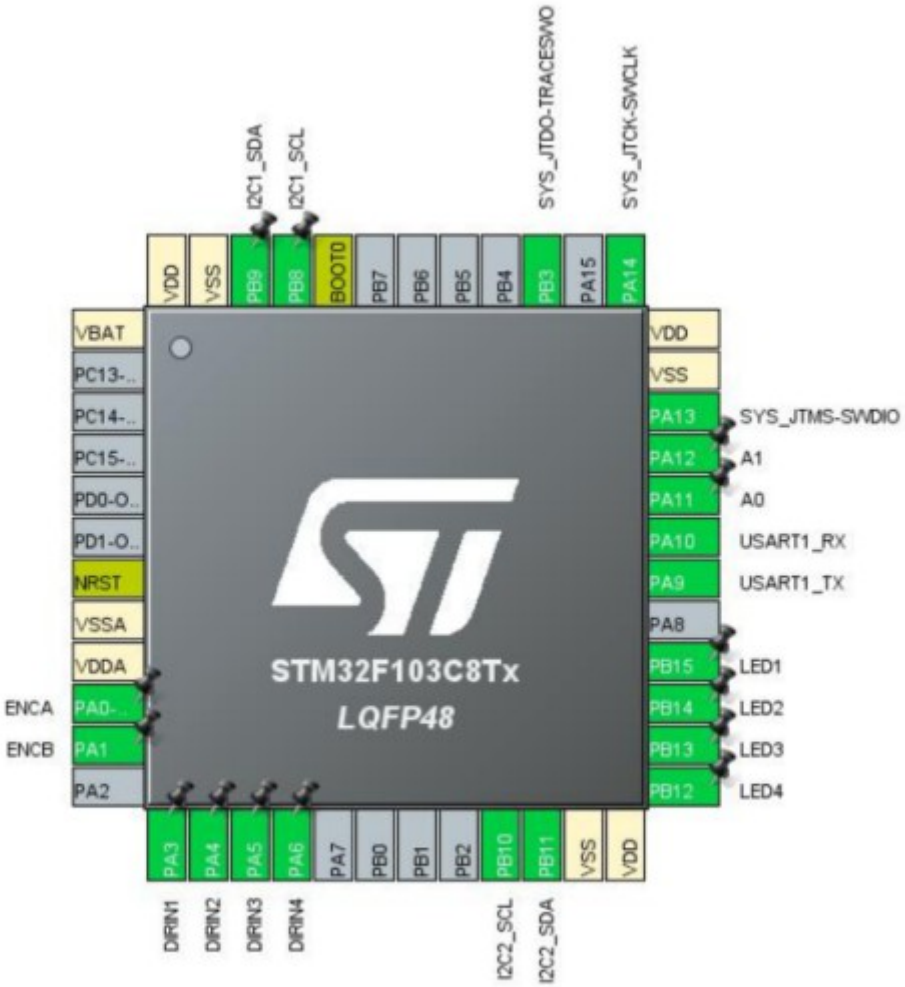
Deberá ajustarse a las particularidades de cada proyecto.

Entradas y salidas físicas



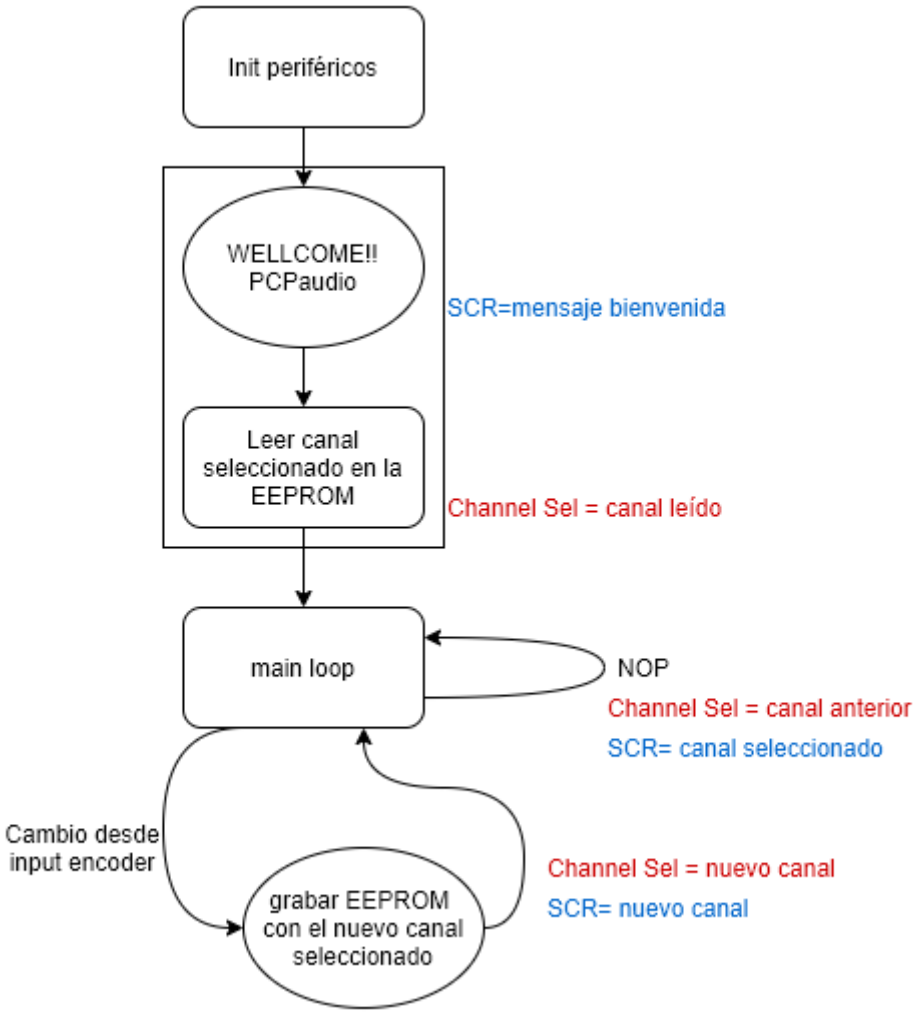


Asignación de pines del microcontrolador



Pin Number LQFP48	Pin Name (function after reset)	Pin Type	Alternate Function(s)	Label
1	VBAT	Power		
7	NRST	Reset		
8	VSSA	Power		
9	VDDA	Power		
10	PA0-WKUP	I/O	GPIO_EXTI0	ENCA
11	PA1 *	I/O	GPIO_Input	ENCB
13	PA3 *	I/O	GPIO_Input	DIRIN1
14	PA4 *	I/O	GPIO_Input	DIRIN2
15	PA5 *	I/O	GPIO_Input	DIRIN3
16	PA6 *	I/O	GPIO_Input	DIRIN4
21	PB10	I/O	I2C2_SCL	
22	PB11	I/O	I2C2_SDA	
23	VSS	Power		
24	VDD	Power		
25	PB12 *	I/O	GPIO_Output	LED4
26	PB13 *	I/O	GPIO_Output	LED3
27	PB14 *	I/O	GPIO_Output	LED2
28	PB15 *	I/O	GPIO_Output	LED1
30	PA9	I/O	USART1_TX	
31	PA10	I/O	USART1_RX	
32	PA11 *	I/O	GPIO_Output	A0
33	PA12 *	I/O	GPIO_Output	A1
34	PA13	I/O	SYS_JTMS-SWDIO	
35	VSS	Power		
36	VDD	Power		
37	PA14	I/O	SYS_JTCK-SWCLK	
39	PB3	I/O	SYS_JTDO-TRACESWO	
44	BOOT0	Boot		
45	PB8	I/O	I2C1_SCL	
46	PB9	I/O	I2C1_SDA	
47	VSS	Power		
48	VDD	Power		

Máquina de estados



Esquema de entradas y salidas

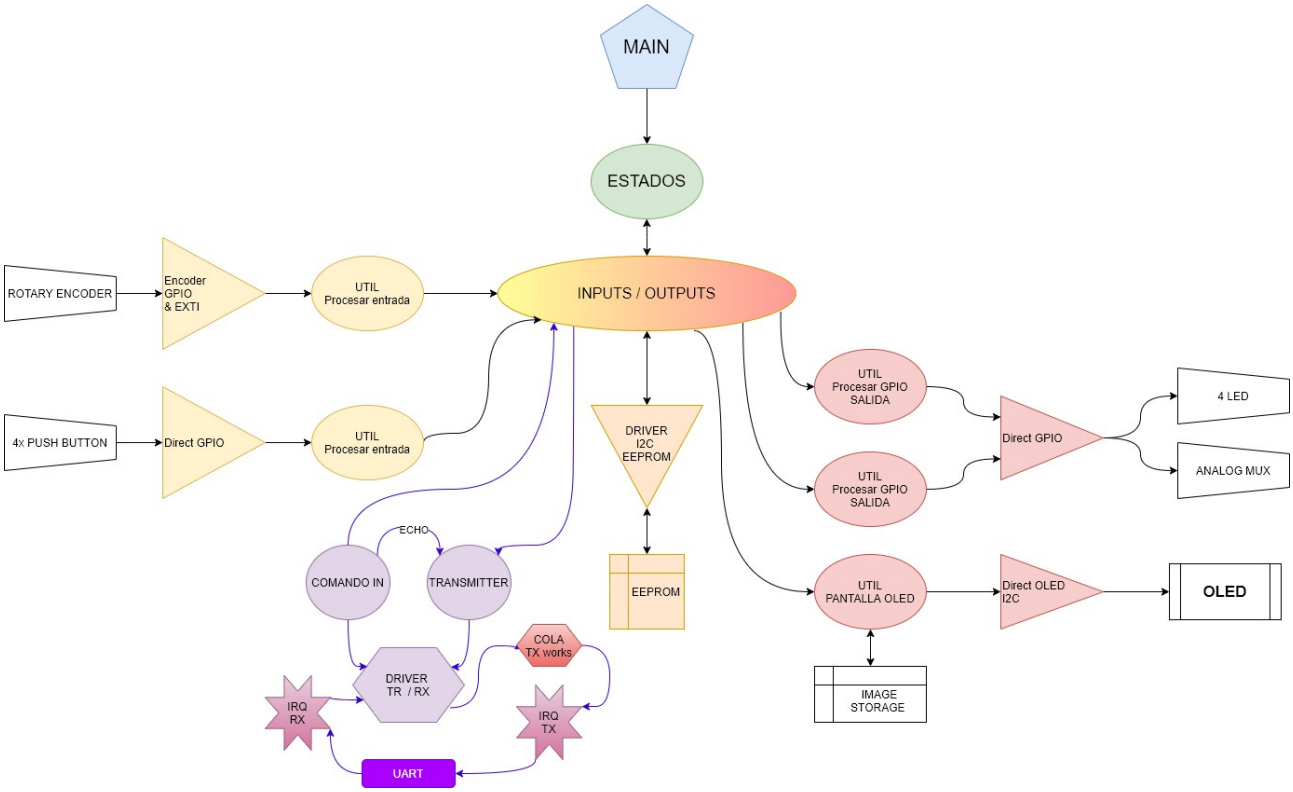
Este esquema de entradas y salidas es muy similar al de AC input, de ahí se deben tomar todas las referencias posibles. Se copiarán los módulos:

Drivers GPIN / Gpout – se puede pensar en unificar

TODAS LAS COMUNICACIONES- Comando in, Transmitter, Driver TXRX, cola TX.

Inputs, Outputs, Estados.

Jerarquía de drivers/útiles



Este es el esquema jerárquico que representa los módulos del programa: drivers y útiles, inputs, input/ouputs y outputs.

Como el bloque de comunicaciones se va complicando, en el futuro se sustituirá por una bloque conceptual de "comunicaciones".

Estructura de datos de entradas y salidas

En esta arquitectura ya están unificados inputs y outputs.

Estructura básica de un objeto IO

DATOS:

.dat

Aqui se almacenan todos los datos del útil.

Se crea con un constructor

IOs[IN_ENCODER].dat = (Encoder_t*)NewEncoder();

.size

uint8_t , para cuando se use un vector, definir su tamaño en [0]

MÉTODOS:

Estos son los métodos estándar

.init

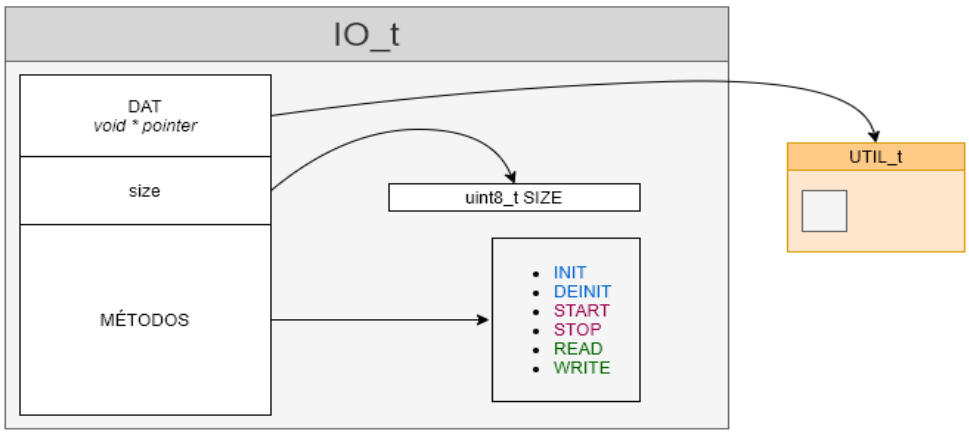
.deinit

.start

.stop

.write

.read



Arrays de Input/outputs

En esta versión de la arquitectura de software los objetos input y output están agrupados sólo en objetos IO.

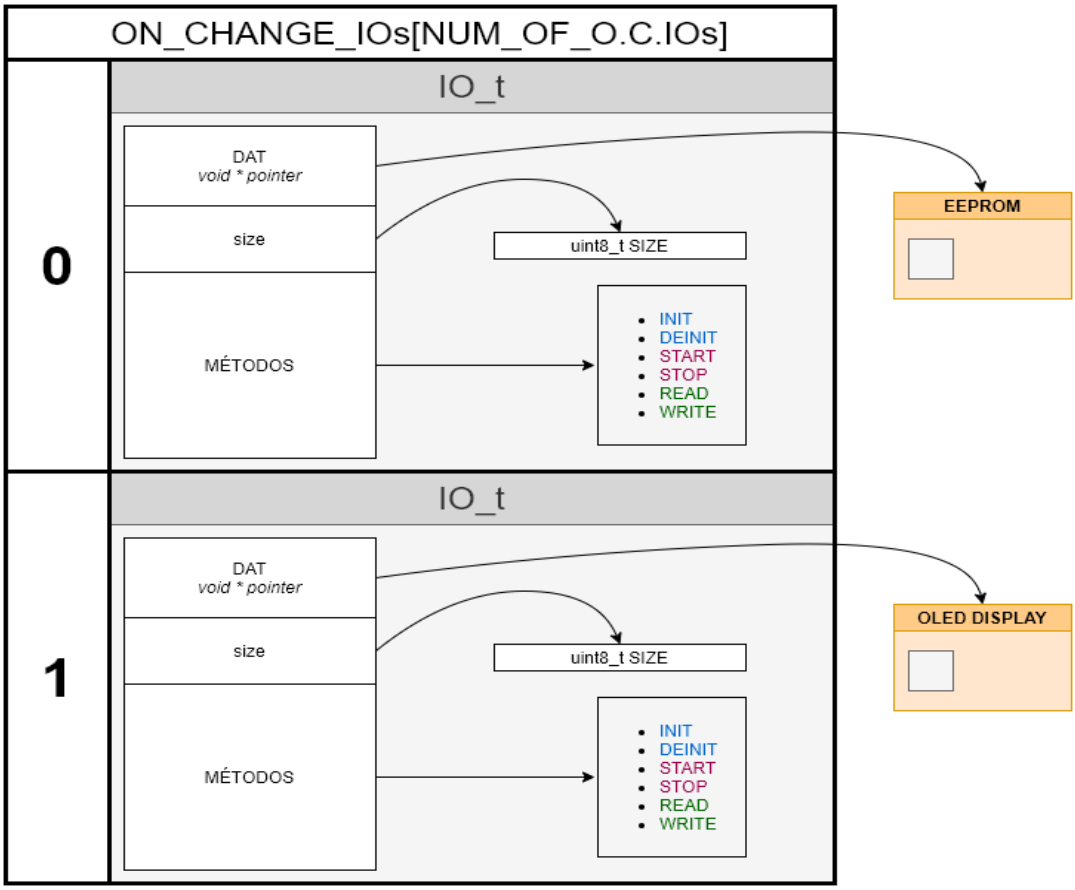
Se distingue entre dos grupos, argupados en dos ***array*** de objetos IO, RegIO y OnChange_IO.

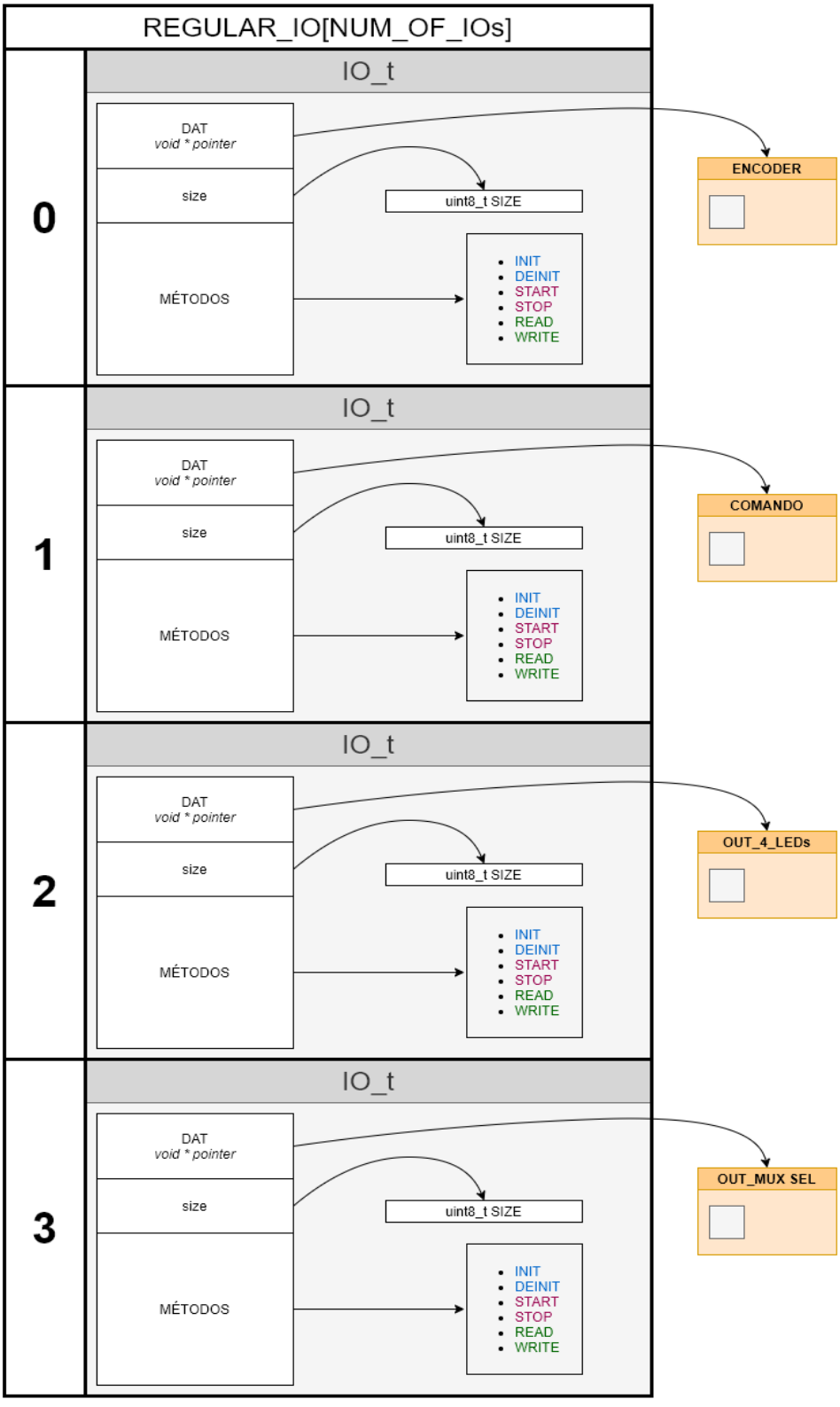
RegIO son los objetos regulares:

- Encoder
- In Comando
- Out Leds
- Out Channel Selection

OnChange_IO son los objetos que sólo hay que tratar si se produce un cambio en el estado

- OLED
- FLASH (en realidad es una EEPROM)




REGULAR_IO[NUM_OF_IOs]
0
IO_t
DAT
void * pointer
size
MÉTODOS
uint8_t SIZE
INIT
DEINIT
START
STOP
READ
WRITE
ENCODER
1
IO_t
DAT
void * pointer
size
MÉTODOS
uint8_t SIZE
INIT
DEINIT
START
STOP
READ
WRITE
COMANDO
2
IO_t
DAT
void * pointer
size
MÉTODOS
uint8_t SIZE
INIT
DEINIT
START
STOP
READ
WRITE
OUT_4_LEDs
3
IO_t
DAT
void * pointer
size
MÉTODOS
uint8_t SIZE
INIT
DEINIT
START
STOP
READ
WRITE
OUT_MUX SEL

Inicialización de los arrays REG_IO y OnChange_IO

En el constructor de los arrays de IO_t's, se llama a las funciones constructoras de cada objeto del array, y se asignan sus métodos:

```
IOs[OUT_LEDs].dat    = (Util_4_LEDs_t *)NewUtil_4_LED();
IOs[OUT_LEDs].init   = Out4LEDs_Init;
IOs[OUT_LEDs].deinit = Void_IO_Function;
IOs[OUT_LEDs].start  = Void_IO_Function;
IOs[OUT_LEDs].stop   = Void_IO_Function;
IOs[OUT_LEDs].write  = Out4LEDs_Write;
IOs[OUT_LEDs].read   = Void_IO_Function;
```

Para los métodos no definidos, se emplea la función Void_IO_Function:

```
bool Void_IO_Function(IO_t * in){
    return true;}
```

Como se puede ver, no hace nada y cumple con el ***typefef*** de los métodos de un IO_t

Asignación de variables para las IRQ

Un caso especial supone la asignación de las variables reservadas para las rutinas de servicio de interrupción. Por ahora este es el mejor lugar para asignarlas:

```
IOs[IN_ENCODER].dat    = (Encoder_t*)NewEncoder();
IOs[IN_ENCODER].init   = Encoder_Init;
IOs[IN_ENCODER].deinit = Void_IO_Function;
IOs[IN_ENCODER].start  = Void_IO_Function;
IOs[IN_ENCODER].stop   = Void_IO_Function;
IOs[IN_ENCODER].write  = Void_IO_Function;
IOs[IN_ENCODER].read   = Encoder_Read;
//Asignaciones de IRQ
EncoderInt = IOs[IN_ENCODER].dat;

IOs[IN_COMANDO].dat    = (UtilComm_IN_t *)NewUtilComm_IN();
IOs[IN_COMANDO].init   = Comando_Init;
IOs[IN_COMANDO].deinit = Comando_Write;
```

```
IOs[IN_COMANDO].start  = Comando_Start;
IOs[IN_COMANDO].stop   = Comando_Stop;
IOs[IN_COMANDO].write  = Comando_Write;
IOs[IN_COMANDO].read   = Comando_Read;
//Asignaciones de IRQ
        RXCommInt = Ios[IN_COMANDO].dat;
```

Funciones asignadas a los métodos:

Por ahora, hay que definir las funciones que se asignan al método de cada objeto IO del array. Se hace en el propio archivo IO.c,

Todos las funciones tienen esta estructura. La única diferencia entre ellas es el cast del tipo y la operación a la que se refieren:

```
////////////////////////////////////////
// MUX selector A0,A1
// Init, Write
bool OutMux_Init(struct IO_struct * In){
        return ( (Out_Sel_Ch_t *)In )-> init ( (Out_Sel_Ch_t *)In );
}
bool OutMux_Write(struct IO_struct * In){
        return ( (Out_Sel_Ch_t *)In )-> write ( (Out_Sel_Ch_t *)In );
}
////////////////////////////////////////
// EEPROM
// Init, Read, Write
bool EEPROM_Init(struct IO_struct * In){
        return ((E2PROM_t *)In)-> init ((E2PROM_t *)In);
}
```

Se podría pensar en una forma de **simplificar** estas funciones, porque **tienen una estructura común**, pero a la vez es **necesario** que sea **eficiente**, que no tengan "If's" eternos o switches con muchos casos. Sería más interesante un indexado directo por ejemplo

Super objeto con métodos

Existe un superobjeto IO_t que nos permitirá llamar a los métodos desde el main, y que permite ejecutar en bucle for un método concreto en todas las los

Por ejemplo: *IO->read(RegIO);*

```
IO_t * NewGenIO(void){
        static IO_t IO_Main={
                .dat  = NULL,
                .size = 0,
                .init   = initIOs,
                .deinit = deinitIOs,
                .start  = startIOs,
                .stop   = stopIOs,
                .write  = writeIOs,
                .read   = readIOs
        };
                return &IO_Main;
}
```

Funciones de los métodos del superobjeto:

Ya que todas las funciones tienen una estructura similar, se define una macro:

```
#define __OP(operation)   bool resultado= true;\
                          for (int i = 0; i < (&IO_Array[0])->size ; i++)\
                          {resultado &= IO_Array[i].operation( IO_Array[i].dat);}\
                          return resultado;\
```

Luego, las funciones se definen así:

```
bool initIOs(IO_t *IO_Array){
        __OP(init)}
bool deinitIOs(IO_t *IO_Array){
        __OP(deinit)}
```

```
bool startIOs(IO_t *IO_Array){
        __OP(start)}
....
```

Gestión de la máquina de estados

Módulo de estados:

En el archivo variable_estado.c/h se almacena un *enum* para definir los estados. Se genera la variable global (extern) **estado**.

Tipo de datos estado:

```
typedef enum EnumEstados {
        error = 0,
        welcome = 1,
        Ch1  = 2,
        Ch2  = 3,
        Ch3  = 4,
        Ch4  = 5
} Estados_t;
```

Función Info_a_Salidas

escribe la info a los handler de las salidas, en función del estado

IO_Array[OUT_LEDs].dat->Channel = *qué debe tener la salida de Out Leds*

IO_Array[OUT_CH_SEL].dat->Channel = *qué debe tener la salida de Out Ch Sel*

...

En función del estado, deberán tener uno u otro

```
switch ( estado ){
        case  welcome: [...]
        case Ch1:[...]
```

...

Posteriormente, se usa el método **write** del superobjeto IO

IO->write(RegIO);

IO->write(OnChange_IO);

Función ProcesarEstado

bool ProcesarEstado(IO_t *IO_Array, IO_t * OnChangeIO_Array){

Esta función elige qué se lee y cómo se cambia de estado, en función del estado actual

```
switch ( estado ){

        case error:
                while(1){} // bucle infinito y que nos reinicie el watchdog
        case welcome:
                if (EEaux->read( EEaux ) ){
                ,,
        case Ch1:
                // mirar si algo es distinto de como esta ahora
                if ( Check_Encoder( &EncAux->status, Ch2, Ch4) ){break;}
                Check_Ch_ComStat(CommAux->ComStat);
                break;
        ...
 Info_a_Salidas (IO_Array, OnChangeIO_Array);
```

Funciones auxiliares para comprobar eventos de entrada:

- Check_Encoder
- Check_Ch_ComStat

Estas funciones ayudan aprocesar los eventos de entrada y el cambio de estado

Función Check_Encoder

Esta función mira si hay alguna operación de **UP/DOWN** pendiente en el **encoder**.

```
bool Check_Encoder( estadosEncoder_t *status, Estados_t ChUp, Estados_t ChDown){
        if ( *status != nop){  // Si hay algún evento se mira si es up o down
```

```
if  ( *status == up){
        *status= nop;
        estado = ChUp;
else es down...
```

Función Check_Ch_ComStat

Esta función mira si hay algún flag en el array de **flags** de **comandos** para cambiar a alguno de los canales:

```
bool Check_Ch_ComStat (bool * ComStat){
    if ( ComStat[Ch1] == true) {
        ComStat[Ch1] = false;
        estado = Ch1;
        return true;
    ...
```

Aqui se cometió un error al no indexar el vector ComStat[n] con un índice válido.

Útil del Encoder

El útil del Encoder se encarga de gestionar el cambio de canal producido desde el HMI encoder rotativo.

Opera por interrupción.

Se basa en el driver de GPIOs

Tipos de datos

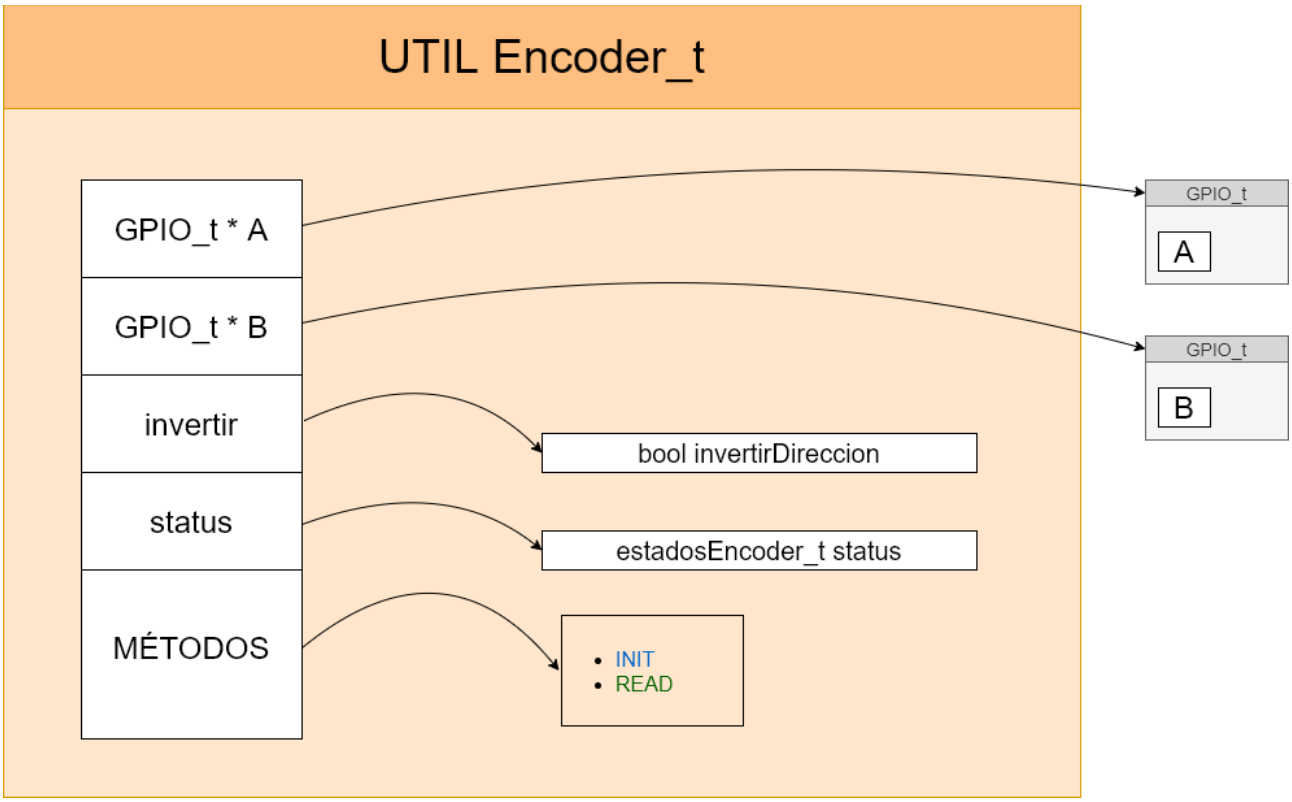
Se basa en la clase Encoder_t

```
typedef struct EncoderStruct{
        //data
        bool invertirDireccion;
        estadosEncoder_t status;
        GPIO_t *A;
        GPIO_t *B;
        //métodos
        Encoder_Handler_t      init;
        Encoder_Read_Handler_t read;
} Encoder_t;
```

Para que la rutina de servicio de interrupción pueda operar, se genera una variable extern, que se inicializará en el módulo Ios, en el constructor del array de IO_ regular:

```
extern Encoder_t * EncoderInt;
```

Esta variable es un puntero que apunta al objeto *Encoder_t* generado.



Funciones asociadas a la clase:

Existen tres funciones, una de ellas queda fuera porque es la rutina de servicio de interrupción.

Rutina de servicio de interrupción.

La definimos en el propio fichero "Encoder.c", y la borramos de "stm32f1_it.c"

Esta función además incorpora un *debouncer* por software para impedir falsos disparos.

```
void EXTI0_IRQHandler(void) {
    // esta función es un debouncer por software.
    // Capta el Tick, y si no han pasado MIN_TIME milisegundos desde el anterior, no responde
    #define MIN_TIME 10
    static uint32_t FormerTick; // Debe mantener el valor e inicializarse a cero
    if ( HAL_GetTick() - FormerTick > MIN_TIME){
        //## código del encoder - pasado debouncing criteria
            // Cuando llegue la interrupción es que se ha producido un flanco ascendente en el pin ENC-A
            // hay que leer ENC- B para detectar la dirección de giro. Si está Low es anticlockwise
```

```
                    // y si está up es clockwise
                    bool resultado;
                    resultado = EncoderInt->B->read(EncoderInt->B);
                    if (resultado ^ EncoderInt->invertirDireccion) {// Se hace un XOR con Invertir Dir.
                            EncoderInt->status = down;
                    }
                    else  {  EncoderInt->status = up;  }
                FormerTick = HAL_GetTick();// Refrescamos el tick para el siguiente ciclo, para el debouncer
        }
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}
```

Función Init:

La función init tiene la relevancia de que los valores se definen aqui, no en el constructor como se había hecho antes.

```
bool Init_ENCODER ( Encoder_t * Encoder){

        Encoder->A= NewGPPin();// Creamos los objetos GPpin
        Encoder->B= NewGPPin();

        Encoder->A->pin  = ENCA_Pin;  // Asignamos pin y puerto del A
        Encoder->A->port = ENCA_GPIO_Port;

        Encoder->B->pin  = ENCB_Pin;  // Asignamos pin y puerto del B
        Encoder->B->port = ENCB_GPIO_Port;
        Encoder->invertirDireccion = false; // Asignamos si se invierte o no la función UP/DOWN en función de la dirección de giro.
        Encoder->A->init();  // Inicializamos los GPIO. Esta función, que está en el driver GPIO, llama a GPIO_Init_MX de la librería HAL.
        return true;
}
```

función READ:

Esta es una función dummie, no hace más que extraer la información que tiene el propio objeto, porque **se lee por interrupción**.

```
estadosEncoder_t Read_ENCODER( Encoder_t * Encoder){
        // Se lee por interrupción, en EXTI0_IRQn
        return Encoder->status;
}
```

Útil de Output 4 LEDs y MUX_Channel_Selector

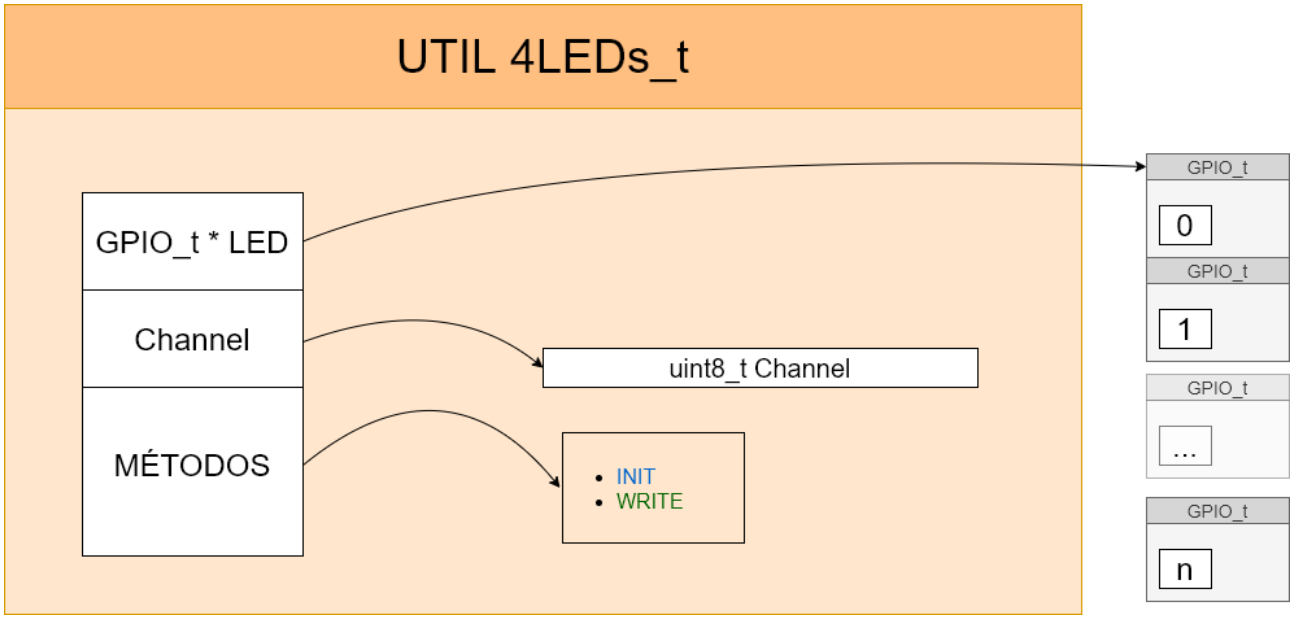
Tipos de datos

Estos útiles se basa en una clase, una para cada uno. La estructura es muy similar, pero responde a las necesidades de cada tipo de salida.

4 LEDs por ejemplo es una salida de 4 pines demultiplexada, donde sólo se enciende uno de los LEDs, mientras que MUX_Selector se compone de dos pines que están codificados en binario.

El tipo es muy básico, sólo contiene un array de GPIOs con la info de los pines correspondientes y el canal o LED seleccionado.

```
typedef struct Util_4_LEDsstruct{
        //data
        GPIO_t * LED[NUM_OUTPUTLEDS];
        LedCH_t / uint8_t Channel;
        //métodos
        Util_4_LEDs_Handler_t init;
        Util_4_LEDs_Handler_t write;
} Util_4_LEDs_t;
```



Funciones asociadas a los métodos:

Funciones INIT.

En ambos casos, los Init consisten en llamar n veces al constructor para crear los GPIO_t necesarios:

```
bool Util_4_LEDs_Handler_init ( Util_4_LEDs_t * U4LEDs){
        for (int i=0; i<NUM_OUTPUTLEDS;i++)  {   U4LEDs->LED[i] = NewGPPin();    }
```

Inicializar en bajo nivel (llamar a HAL_GPIO_MX())

```
        U4LEDs->LED[0]->init();
```

Y luego asignar la información del pin y el puerto a estos objetos:

```
        U4LEDs->LED[0]->pin = LED1_Pin;
        U4LEDs->LED[0]->port = LED1_GPIO_Port;
        ...
```

Por último se escriben los estados en el puerto: * ¿Esto está bien?

```
        resultado &= U4LEDs->write(U4LEDs);
        return resultado;
}
```

Funciones Write:

Estas funciones asignan primero el estado de cada bit en función del estado del útil (Channel):

```
bool Util_4_LEDs_Handler_write ( Util_4_LEDs_t * U4LEDs){
        if (U4LEDs->Channel == LED_CH1){
                U4LEDs->LED[0]->status = true;
                U4LEDs->LED[1]->status = false;
                U4LEDs->LED[2]->status = false;
                U4LEDs->LED[3]->status = false;
```

Y por último se ejecuta la escritura llamando al método write de los objetos GPIO_t:

```
        bool resultado = true;
        for (int i=0; i<NUM_CH_MUX;i++){
                resultado &= ChSel->MX[i]->write(ChSel->MX[i]);
        }
        return resultado;
```

Útil y driver de EEPROM

Driver:

Este driver se encarga de aspectos de más bajo nivel como la escrtura en un tipo u otro de bus, el address si es un I2C, etc..

Tipos de datos

En la clase en la que se basa el driver se almacena el handler del I2C, su dirección I2C y un parámetro de la transmisión, el timeout.

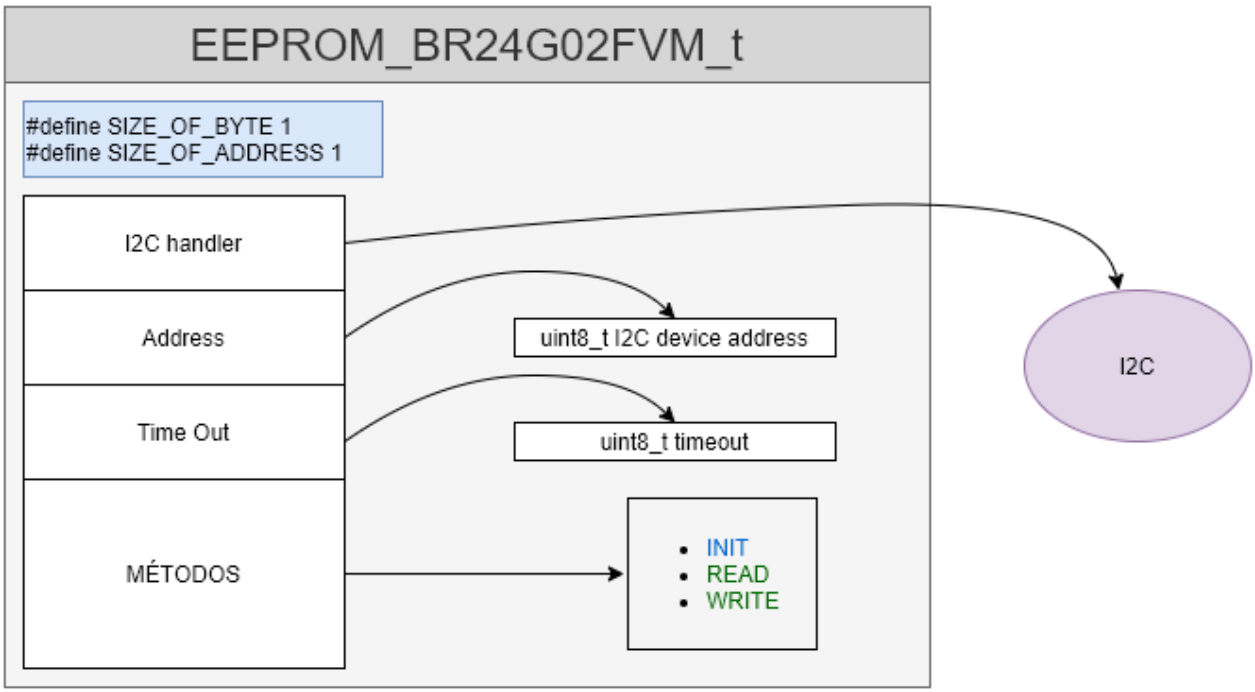
Contiene tres métodos: init, read y write.

```
typedef bool (*EEpromInitHandler_t)(struct Struct_EEPROMBR24G02FVM * EEprom);
typedef bool (*EEpromRWbyteHandler_t) (struct Struct_EEPROMBR24G02FVM * EEprom, uint16_t * address, uint8_t * data);

typedef struct Struct_EEPROMBR24G02FVM{
        //config data
```

```
    I2C_HandleTypeDef *hi2c;
    uint8_t I2CAddress;
    uint8_t TimeOut;
    // métodos:
    EEpromInitHandler_t   init;
    EEpromRWbyteHandler_t read;
    EEpromRWbyteHandler_t write;

} EEPROMBR24G02FVM_t;
```



Funciones asociadas a los métodos:

Init

En esta función se inicializa el periférico I2C, y se hace un "ping" al dispositivo para comprobar que se comunica correctamente con él, mediante la instrucción *HAL_I2C_IsDeviceReady*.

```
bool EEPROM_BR24G02FVM_Init (EEPROM_BR24G02FVM_t * EEprom){
    MX_I2C1_Init();
    if (HAL_OK == HAL_I2C_IsDeviceReady (
```

```
            EEprom->hi2c, EEprom->I2CAddress, NumOfRetry, EEprom->TimeOut))
        { return true;}
        else {return false;}
}
```

Read

Read recibe dos parámetros:

- uint16_t * address – La dirección del registro interno de la EEPROM que se quiere leer
- uint8_t * data – el Byte donde se guardará lo que se lea

Se basa en leer un sólo bit mediante la instrucción *HAL_I2C_Mem_Read*:

```
if (HAL_OK == HAL_I2C_Mem_Read (
                EEprom->hi2c,
                EEprom->I2CAddress ,
                *address,
                SIZE_OF_ADDRESS ,
                 data, SIZE_OF_BYTE,
                EEprom->TimeOut))
        { return true;}
```

Write

Write, al igual que Read recibe dos parámetros:

- uint16_t * address – La dirección del registro interno de la EEPROM donde se quiere escribir
- uint8_t * data – la dirección del dato que se desea guardar en la EEPROM

Se basa en *HAL_I2C_Mem_Write,* y se añade un retardo de 5ms después para garantizar que se escribe correctamente. Esto es algo que se puso sobre la marcha, no se pudo comprobar si era el mínimo tiempo admisible o si tenía otra causa:

```
        if (HAL_OK == HAL_I2C_Mem_Write (
                EEprom->hi2c,
                EEprom->I2CAddress ,
                *address, SIZE_OF_ADDRESS ,
                data,
                SIZE_OF_BYTE,
```

```
            EEprom->TimeOut))
        {
            HAL_Delay(5);
            return true;
        }
        else { return false; }
```

Constructor:

Existe un constructor para este tipo de dato. Inicializa el I2C Address, que es una práctica que no debería ser habitual (debería ir en INIT), sea como sea, el útil no tendrá que lidiar con la dirección física de la memoria.

```
EEPROM_BR24G02FVM_t * NewEEPROM_BR24G02FVM(void){
        static EEPROM_BR24G02FVM_t NewEEPROM_BR24G02FVMvar= {
                .hi2c = &hi2c1,
                .I2CAddress = EEPROM_BR24G02FVM_ADDRESS,
                .TimeOut = EETO,
                // métodos:
                .init  = EEPROM_BR24G02FVM_Init,
                .read  = EEPROM_BR24G02FVM_ReadByte,
                .write = EEPROM_BR24G02FVM_WriteByte
        };
        return &NewEEPROM_BR24G02FVMvar;
}
```

Util

El útil es mejorable en el sentido de que podría hacerse una abstracción mayor de los datos que vamos a manejar, no tener que lidiar con la dirección del registro que queremos leer, sino crear un struct, array, o lo que se prefiera, que almacene el dato de una forma más abstracta. Y si es necesario, que las funciones Read y Write descompongan ese dato en bytes.

Tipos de datos

Existe un tipo de dato que almacena el canal, en un tipo enumerado.

```
typedef enum EEPROM_Enum{
```

```
        EEPROM_CH1=0,
        EEPROM_CH2=1,
        EEPROM_CH3=2,
        EEPROM_CH4=3
} EEPROM_CH_t;
```

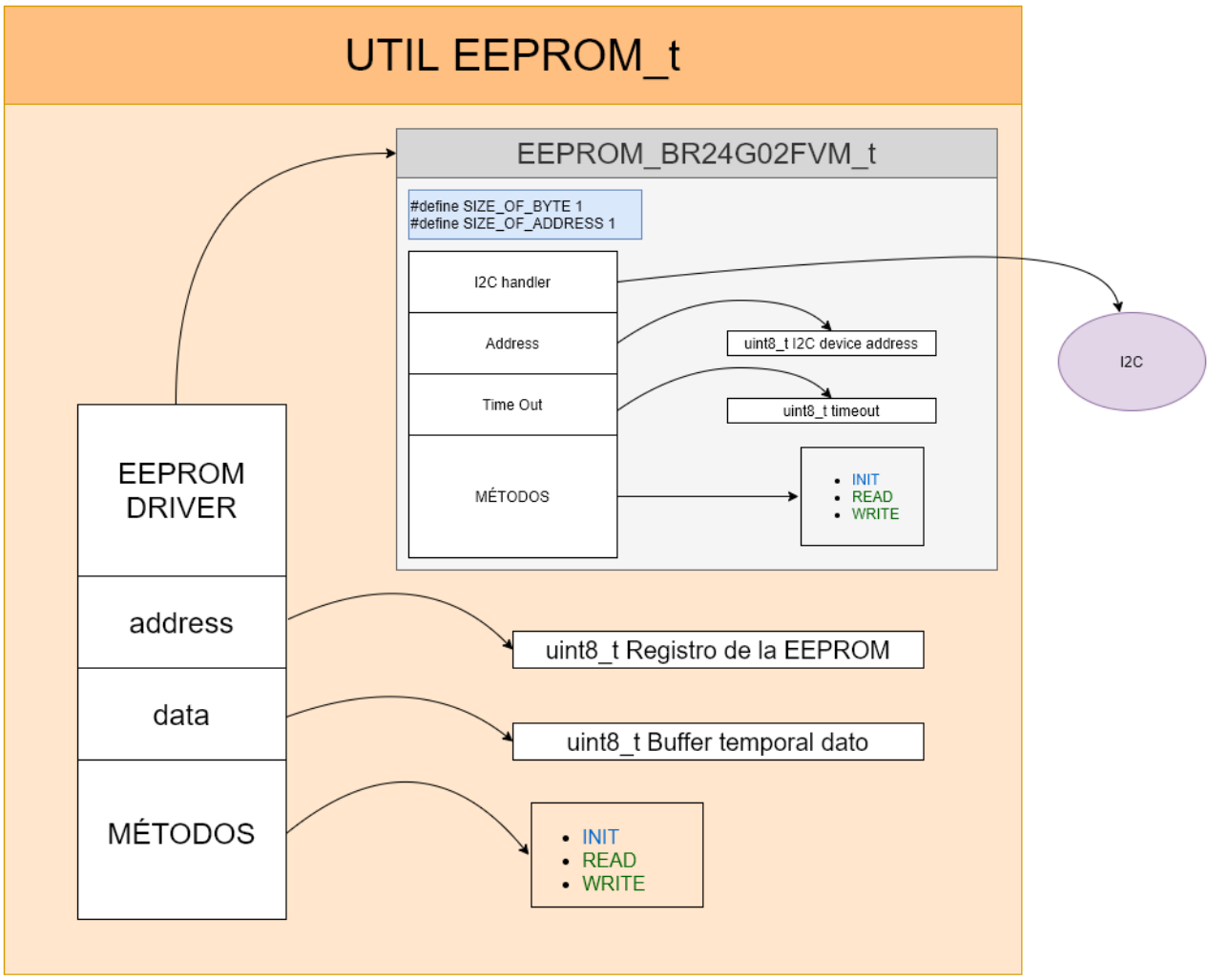
Además, el útil se basa en una clase. Su estructura es muy similar al driver:

```
typedef struct E2PROMstruct{
        //data
        EEPROMBR24G02FVM_t * DrvEEPROM;
        uint8_t data;
        uint8_t RegAddress;
        //métodos
        E2PROM_Handler_t init;
        E2PROM_Handler_t read;
        E2PROM_Handler_t write;
} E2PROM_t;
```

Existen dos bytes que almacenan información clave:

- uint8_t * address – La dirección del registro interno de la EEPROM donde se quiere leer o escribir
- uint8_t * data – la dirección del buffer que almacena el dato que se desea guardar o leer en la EEPROM

Esto es lo que se le pasa como **parámetro** a las funciones *read* y *write* del método.



Útil y driver de OLED Screeen Display

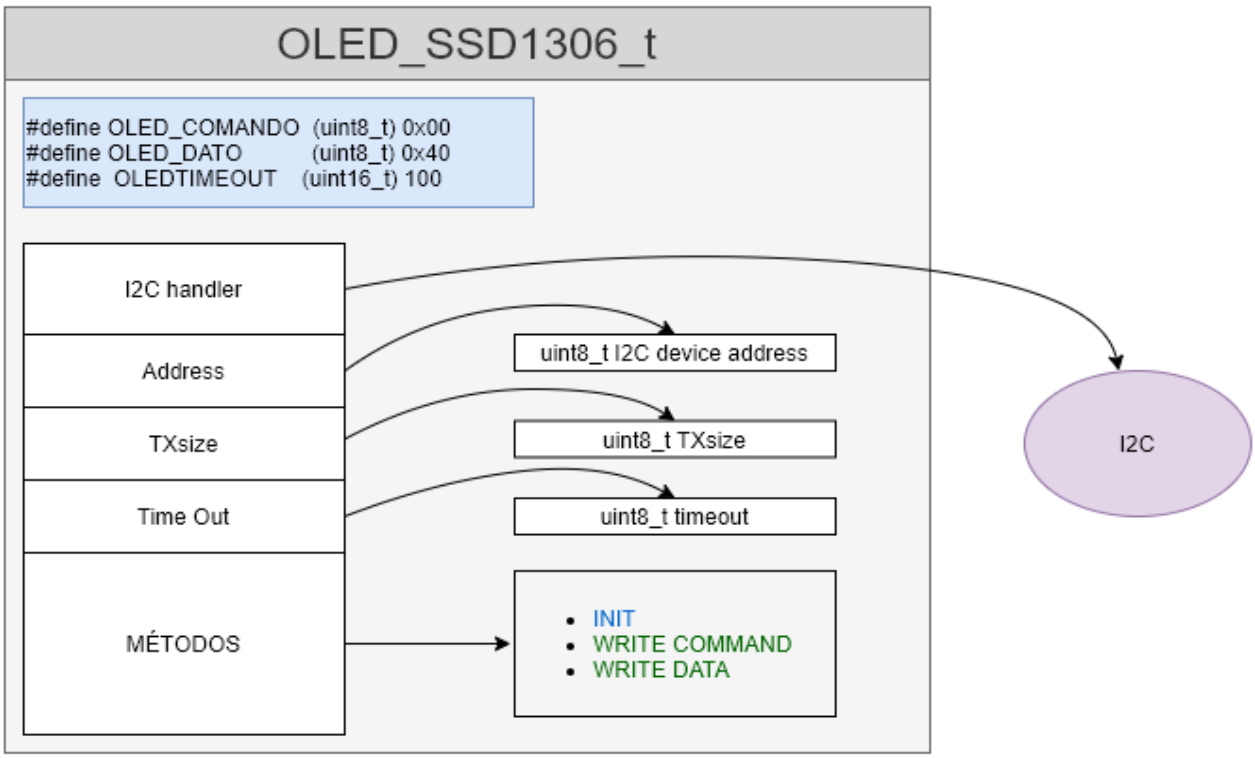
Driver

Este driver usa el puerto I2C para comunicarse con un display SSD1306, de 128x32 píxeles, OLED, monocromo.

Tipos de datos

El driver está contenido en una clase. Es un poco distinta de los drivers convecionales porque en un display se puede escribir dato o comando, siendo órdenes distintas, por eso entre los métodos están *writeCommand* y *writeData*.

```
typedef struct OLEDstruct{
        // Handler del puerto I2c
        I2C_HandleTypeDef * i2c;
        //data
        uint8_t I2Caddress;
        const uint8_t * data;
        uint32_t TXsize;
        uint16_t TimeOut;
        //métodos
        OLED_Handler_t init;
        OLED_Handler_t writeCommand;
        OLED_Handler_t writeData;
} OLED_t;
```



Función init:

La función init debería inicializar el periférico I2C, que es un recurso compartido con la EEPROM. Aqui no se inicializa.

El módulo OLED si que requiere una secuencia de inicialización propia, que se reune en una función separada: *InitialSequence(Oled)*;

Secuencia de inicialización de la pantalla OLED:

Se ha hecho una subrutina que inicializa la pantalla, pero ha habido que tener en cuenta que... había que leerse el datasheet

En cada bloque de instrucción se indica el capítulo del datasheet donde se encuentra la explicación del comando de cada caso.

Se siguen las instrucciones de la pag. 64 de la documentacion "robojax-SSD1306-OLED-manual.pdf".

Ver también "Como conectar una pantalla OLED SSD1306 con Arduino sin librerías _ robologs.pdf".

Para transmitir cada bloque de instrucciones de inicialización se genera un array de comandos (*uint8_t*), que se carga en el puntero de datos a transmitir, se carga su tamaño y se usa la función write del propio driver.

```
// Apagamos la pantalla
Oled->data   = SetOff;
Oled->TXsize = NUM_SetOff;
resultado &= Oled->writeCommand(Oled);
// Configuramos hardware de la pantalla
Oled->data   = configHard;
Oled->TXsize = NUM_CONFIG_HARD;
resultado &= Oled->writeCommand(Oled);
// siguiente...
```

Estos son los bloques de configuración:

1) **Apagar la pantalla:**

```
(0xAE),      //10.1.12 Set Display ON/OFF (AEh/AFh)- Apagar la pantalla
```

2) **Configuración de hardware:**

```
(0xA1),      //Invertir el eje X de pantalla, por si esta girada.
             //Puedes cambiarlo por 0xA0 si necesitas cambiar la orientacion
(0xC8),      //Invertir el eje Y de la pantalla
             //Puedes cambiarlo por 0xC0 si necesitas cambiar la orientacion
(0x81),      //Configurar el contraste
```

(0x7F), //Este valor tiene que estar entre 0x00 (min) y 0xFF (max)

(0xA6), //10.1.10 Set Normal/Inverse Display (A6h/A7h)- Poner la pantalla en modo Normal

(0xD5),(0x80), //Establecer la velocidad del Oscilador

(0x8D),(0x14) //Activar el 'charge pump'

3) **Configuración de la pantalla:**

(0xA8),(0x1F), // 10.1.11 Set Multiplex Ratio (A8h) -
// Establecer el maximo de filas a 0x3F = 63
// es decir, ira de 0 a 63, por tanto tenemos 64 filas de pixeles
// En nuestro caso 1F-31

(0xD3),(0x00), //Poner el offset a 0

(0x40), //Poner el comienzo de linea a 0

(0xDA), //Mapear los pines COM

(0x40), //Al parecer, la unica configuracion que funciona con mi modelo es 0x12, a pesar
// de que en la documentacion dice que hay que poner 0x02
// PCPnote: aqui la única que funciona es esta pero tiene relación con el cmm 0xA8

(0x20), //10.1.3 Set Memory I2Caddressing Mode (20h)

(0x00) //Horizontal I2Caddressing mode (A[1:0]=00b)

4) **Número de columna y página:**

Aqui los comandos se deben escribir de vez, sin cortar la comuniación, si no no funcionará.

(0x21),(0x00),(0x7F), // 10.1.4 Set Column I2Caddress (21h)
// Columnas de 0 a 127 - triple byte command

(0x22),(0x00),(0x03), // 10.1.5 Set Page I2Caddress (22h)
// Paginas de 0 a 3 - triple byte command

(0xB0),(0x00),(0x10) //10.1.13 Set Page Start I2Caddress for
// Page I2Caddressing Mode (B0h~B7h)

5) **Encender la pantalla:**

```
(0xA4),     //10.1.9 Entire Display ON (A4h/A5h) - Este comando ordena al chip
            // que active el output de la pantalla en funcion
            // del contenido almacenado en su GDDRAM
(0xAF)      //10.1.12 Set Display ON/OFF (AEh/AFh)- Encender la pantalla
```

Función Write Command y Write Data

Estas funciones sólo se diferencian en el dato inluído en rojo:

Para trabajar, requiere que se le indique comando o dato, como los de bus paralelo. Y para eso, el primer dato que se le manda es el registro al que va dirigida. Tiene dos, comando =00 y dato =04. Luego la transimisión I2C continúa, metiendo datos o comandos, y hasta que físicamente la transimisió no para, se sigue metiendo datos ahí.

Si se reinicia la transmisión, se debe mandar el registro.

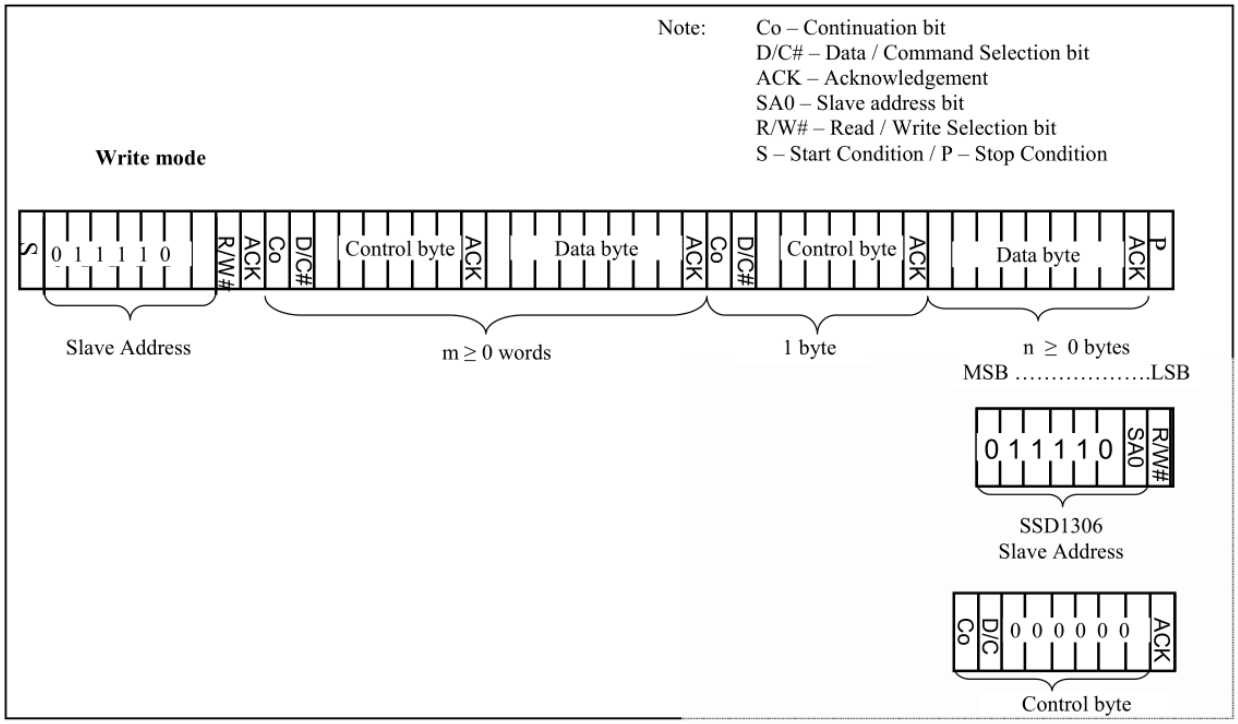
Resumen: usar ***MEM_Write***

```
bool DriverOLEDSSD1306_writeCommand ( OLED_t * Oled){

        HAL_StatusTypeDef res;
        res = HAL_I2C_Mem_Write(
                        Oled->i2c ,
                        Oled->I2Caddress ,
                        OLED_COMANDO / OLED_DATO ,
                        1 ,
                        (uint8_t *)Oled->data ,
                        Oled->TXsize ,
                        Oled->TimeOut
                );
        return (res == HAL_OK )? true:false;
}
#define OLED_COMANDO        (uint8_t) 0x00
#define OLED_DATO           (uint8_t) 0x40
```

Estos defines son el Contro Byte mostrado a continuación, y control byte es el registro dato / comando:

Figure 8-7 : I²C-bus data format



Constructor:

Es importante comentar el address: I2Caddress = **0x78.**

La dirección usual de las pantallas Oled basadas en el chip SSD1306 es 0x3C, pero aqui haciendo un scan sale esto, que es 3C desplazado un bit a la izquierda

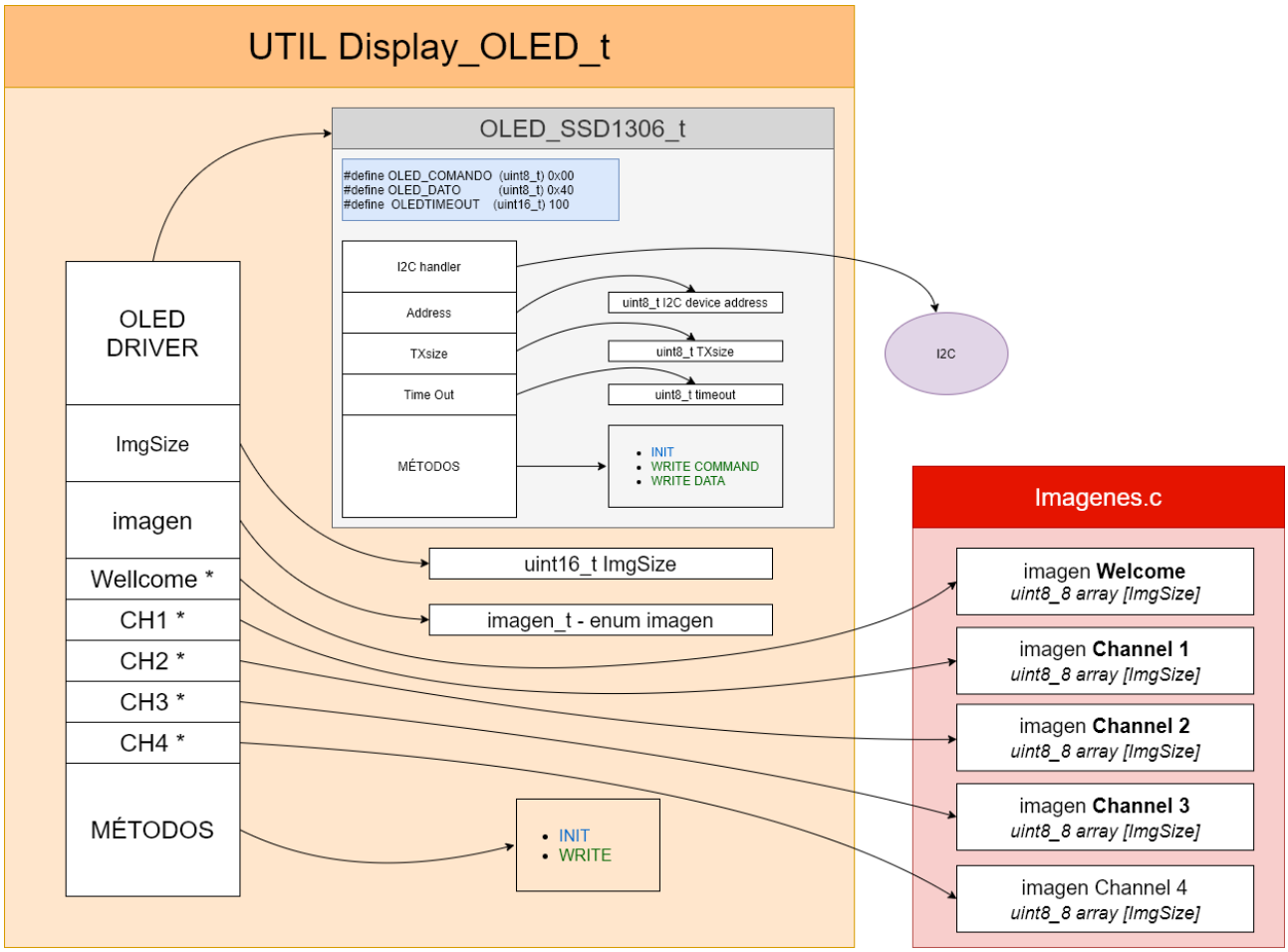
Se trata de un constructor simple, versión 1,0.

```
OLED_t * NewOLED(void){
    static OLED_t NewOLEDvar ={
        //data
        .i2c= & hi2c2,
        .I2Caddress = 0x78,
        .TimeOut = OLEDTIMEOUT,
        //method
        .init  = DriverOLEDSSD1306_init,
        .start = DriverOLEDSSD1306_start,
```

```
        .stop  = DriverOLEDSSD1306_stop,
        .read  = DriverOLEDSSD1306_read,
        .writeCommand = DriverOLEDSSD1306_writeCommand,
        .writeData    = DriverOLEDSSD1306_writeData
    };
    return &NewOLEDvar;
}
```

Util OLED:

El útil del display OLED se basa en una clase que contiene el driver, los métodos y los datos de las imágenes, y en un archivo de imágenes separado, llamado ***imagenes.c***



Tipos de datos:

El útil se basa en esta clase:

```
typedef struct Display_OLEDstruct{
        //Handler:
        OLED_t * Oled;
        //data
        uint16_t ImgSize;
        //punteros a las imágenes
        const uint8_t * welcome;
        const uint8_t * CH1;
        const uint8_t * CH2;
        const uint8_t * CH3;
        const uint8_t * CH4;
        imagen_t imagen;
        //métodos
        Display_OLED_Handler_t init;
        Display_OLED_Handler_t start;   // void function
        Display_OLED_Handler_t stop;   // void function
        Display_OLED_Handler_t read;   // void function
        Display_OLED_Handler_t write;
} Display_OLED_t;
```

imagen es la imagen seleccionada. Se basa en un tipo enumerado para que no pueda salirse del rango. Cada vez que se ejecuta write, se representa esa imagen en la pantalla.

```
typedef enum imagenesEnum{
        OLED_welcome,
        OLED_CH1,
        OLED_CH2,
        OLED_CH3,
        OLED_CH4
}imagen_t;
```

ImgSize es el tamaño de la imagen. En nuestro caso, 128*32 bits = 4096 bits, 512 bytes.

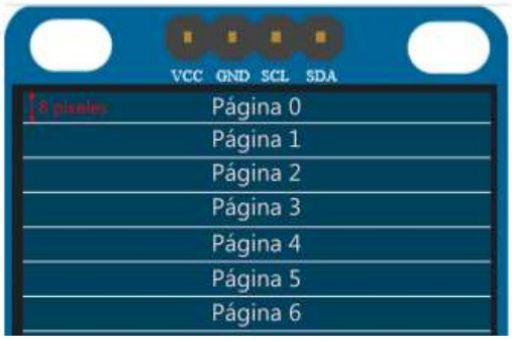
Las imágenes son pesadas en términos de tamaño, y este display es de los más sencillos del mercado.

Tipos de datos: archivo de imágenes

Las imágenes son un array de *uint8_t*, lo declaramos como constante para asegurarnos de que se almacena en la FLASH y no nos desborda la RAM.

```
const uint8_t Imagen_Ch1[IMAGE_SIZE] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0xC0, 0xC0, 0xE0, 0xE0, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0, 0xF0,
0xF0, 0xE0, 0xE0, 0xE0, 0xC0, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0xF0, 0xF0, 0xF0,
0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xF0, 0xF0, 0xF0, 0xF0,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x80, 0xC0, 0xE0, 0xF0, 0xF0, 0xF0,
0xF0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0xF0, 0xFC, 0xFF, 0xFF, 0xFF, 0x1F, 0x07, 0x01, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x01, 0x01, 0x03, 0x0F, 0x1F, 0x1F, 0x1F, 0x1C, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xE0, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0x00, 0x00, 0x00, 0x00, 0x00, 0x30, 0x38, 0x3C, 0x3E, 0x1F, 0x0F, 0x07, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,
0xFF, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
```

Para facilitar la escritura de carácteres de texto las filas se agrupan en ocho páginas, de 8 píxeles de altura cada una.

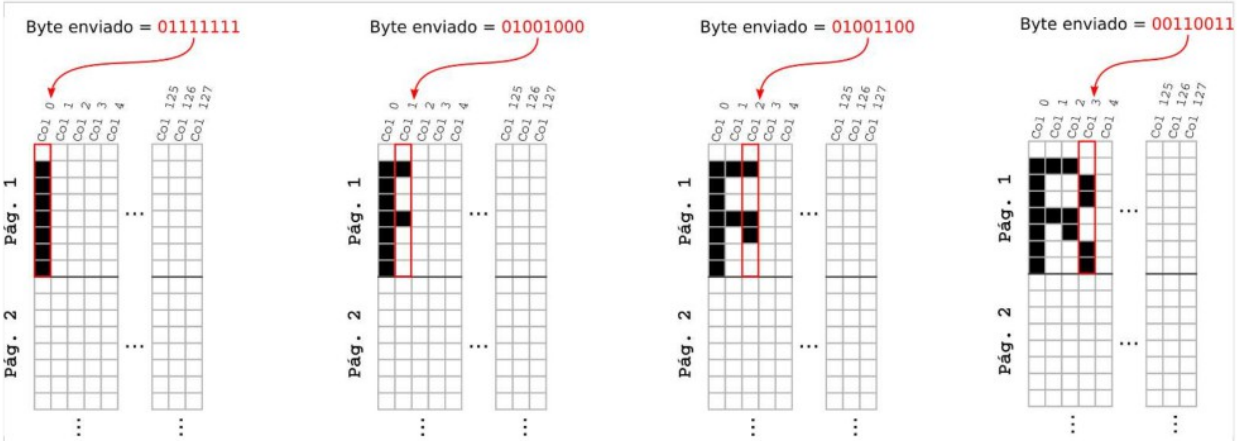


La imagen de la pantalla se modifica enviando *bytes* de datos. Cada **bit** del *byte* de datos se corresponde al estado de un **píxel** de la pantalla. Como la pantalla es monocromática, cada píxel sólo puede tener uno de dos estados: encendido (bit 1) o apapado (bit 0).

El chip tiene varios modos de escritura, es decir, varias formas de colocar los bytes de datos sobre la pantalla. El modo elegido es el de escritura horizontal (Horizontal Adressing Mode). Este modo facilita mucho la escritura de texto.

Al recibir un byte de datos en el modo de escritura horizontal, la pantalla lo colocará en la

primera columna de la primera página e irá llenando las sucesivas columnas con los bytes posteriores.

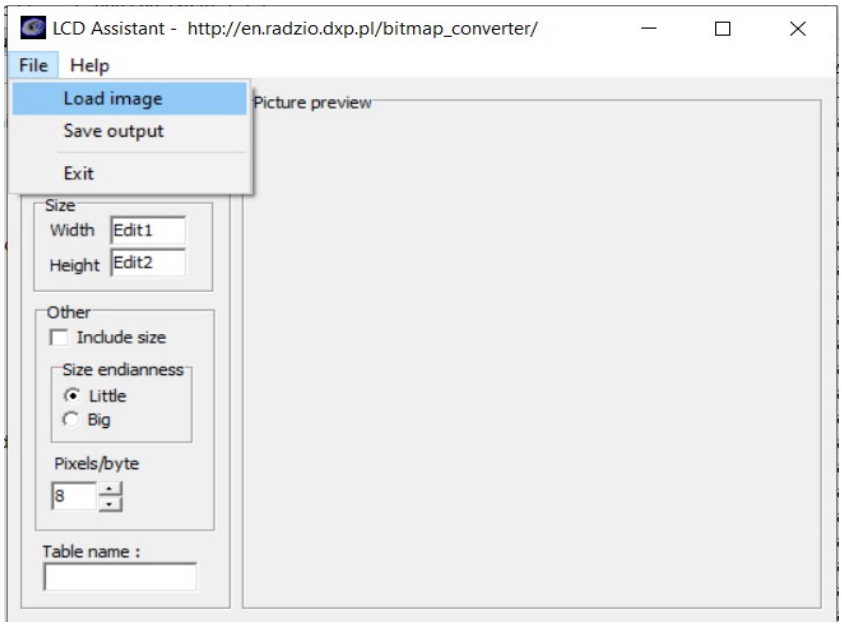


Generación de los arrays de imagen:

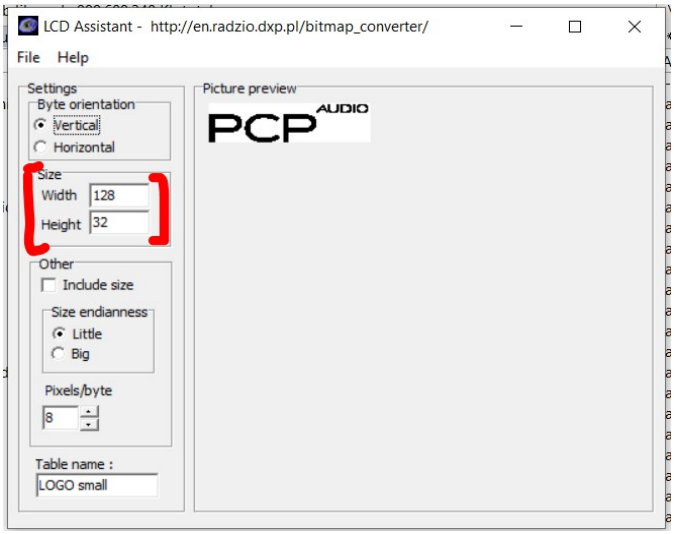
Existe una utilidad para convertir un bitmap a un array de uint8_t, llamada **LCD Converter.**

Es necesario crear un archivo de 128x32 píxeles monocromo, y guardarlo como .BMP.

Tras ejecutarlo, tenemos que abrir el archivo:



Veremos esto:



Grabamos el array generado en file->save output:



Creará un archivo con el nomre que indiquemos, que contendrá los siguientes datos:

```
//-----------------------------------------------------------------------------
// File generated by LCD Assistant
// http://en.radzio.dxp.pl/bitmap_converter/
//-----------------------------------------------------------------------------

const unsigned char LOGO small [] = {
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
...
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

Tendremos que cambiar el const unsigned char por uint8_t y guardarlo en el archivo de imagenes.

Driver de GPIO

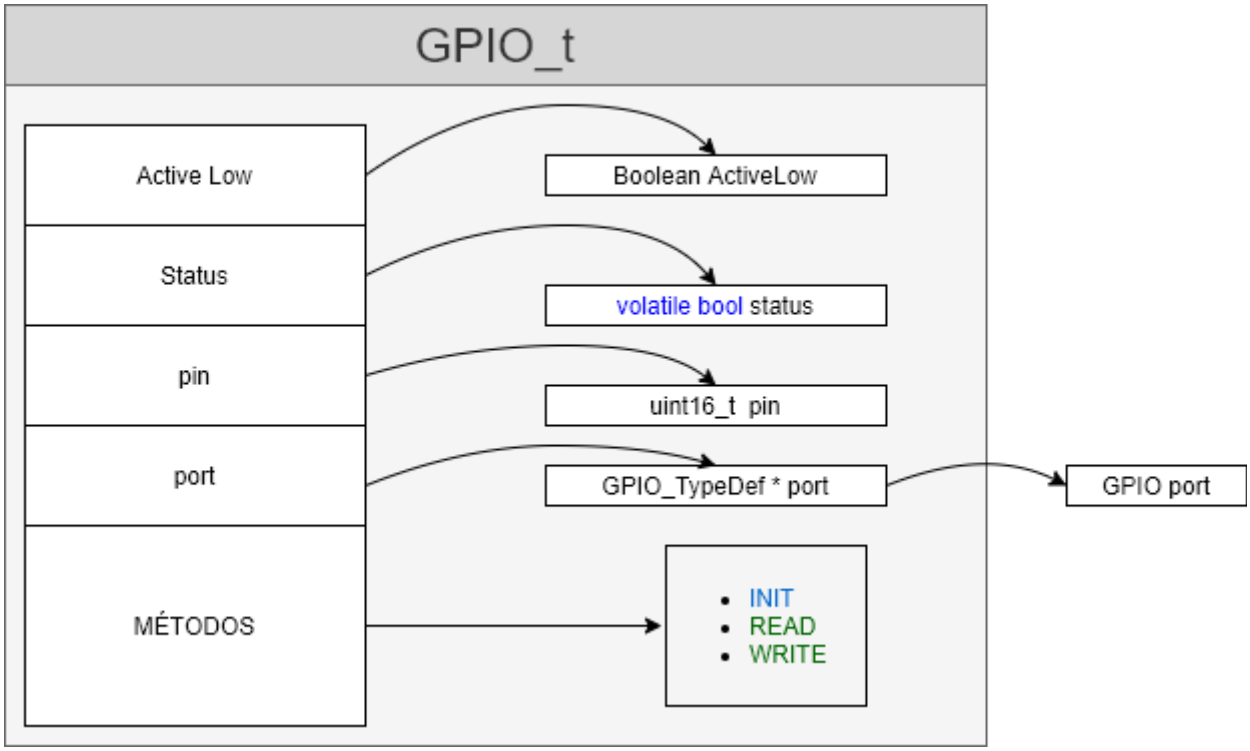
Tipos de datos

Este driver se basa en una clase, que guarda datos relativos al pin:

- Si es activo en bajo
- Su estado (tanto si es input como output).
- El número de pin físico
- El puerto físico

El resto de parámetros se configura en CUBE, y se deja a la librería HAL.

```
typedef struct DriverGPIOstruct {
        // Datos
        bool          ActiveLow;
        volatile bool  status;
        uint16_t       pin;
        GPIO_TypeDef * port;
        // Métodos
        Driver_GPIO_handler_t read;
        Driver_GPIO_handler_t write;
        Driver_GPIO_Init_t    init;
} GPIO_t;
```



Funciones:

En las funciones lo único destacable es que se hace un **XOR** con *Active Low* para que *Status* sea independiente del la representación física de la señal.

Constructor

El constructor si que merece ser explicado en detalle.

Reserva espacio para todos los GPIOs existentes, por eso deben contarse bien.

```
GPIO_t* NewGPPin(){
    // Se crea un array de GPIO_t con todos los GPIO que reservaremos
    // Creamos también un índice estático para llevar la cuenta de
    // cuantos punteros a las posiciones del array se han entregado ya.
    #define NUMERO_DE_GPIOS 8
    static GPIO_t NewGPPinVar[NUMERO_DE_GPIOS];
    static uint8_t index=0;
    // Creamos un booleano que indica si hemos pasado ya
```

```
        // por el bucle de inicialización, donde se asignan los valores por defecto
        static bool init = false;
        if ( ! init ){
                //Asignamos los valores por defecto y los métodos
                for (int i=0; i < NUMERO_DE_GPIOS; i++){
                        NewGPPinVar[i].ActiveLow = false;
                        NewGPPinVar[i].pin  = NULL;
                        NewGPPinVar[i].port = NULL;
                        NewGPPinVar[i].status = false;
                        NewGPPinVar[i].read   = Read_GPIO;
                        NewGPPinVar[i].write  = Write_GPIO;
                        NewGPPinVar[i].init   = Init_GPIO;
                }
                init = true;
        }
        GPIO_t * NewGPPinVar_temp = &NewGPPinVar[index];
        // En cada llamada a la función entregamos punteros a las posiciones
        // del array de GPIO
        if (index < NUMERO_DE_GPIOS){
                index++; //aumentamos el índice para la siguiente entrega
                return NewGPPinVar_temp;
        }
        else{
                return NULL; // Si agotamos las posiciones retornamos NULL.
        }
}
```

COMUNICACIONES

Driver UART unificado

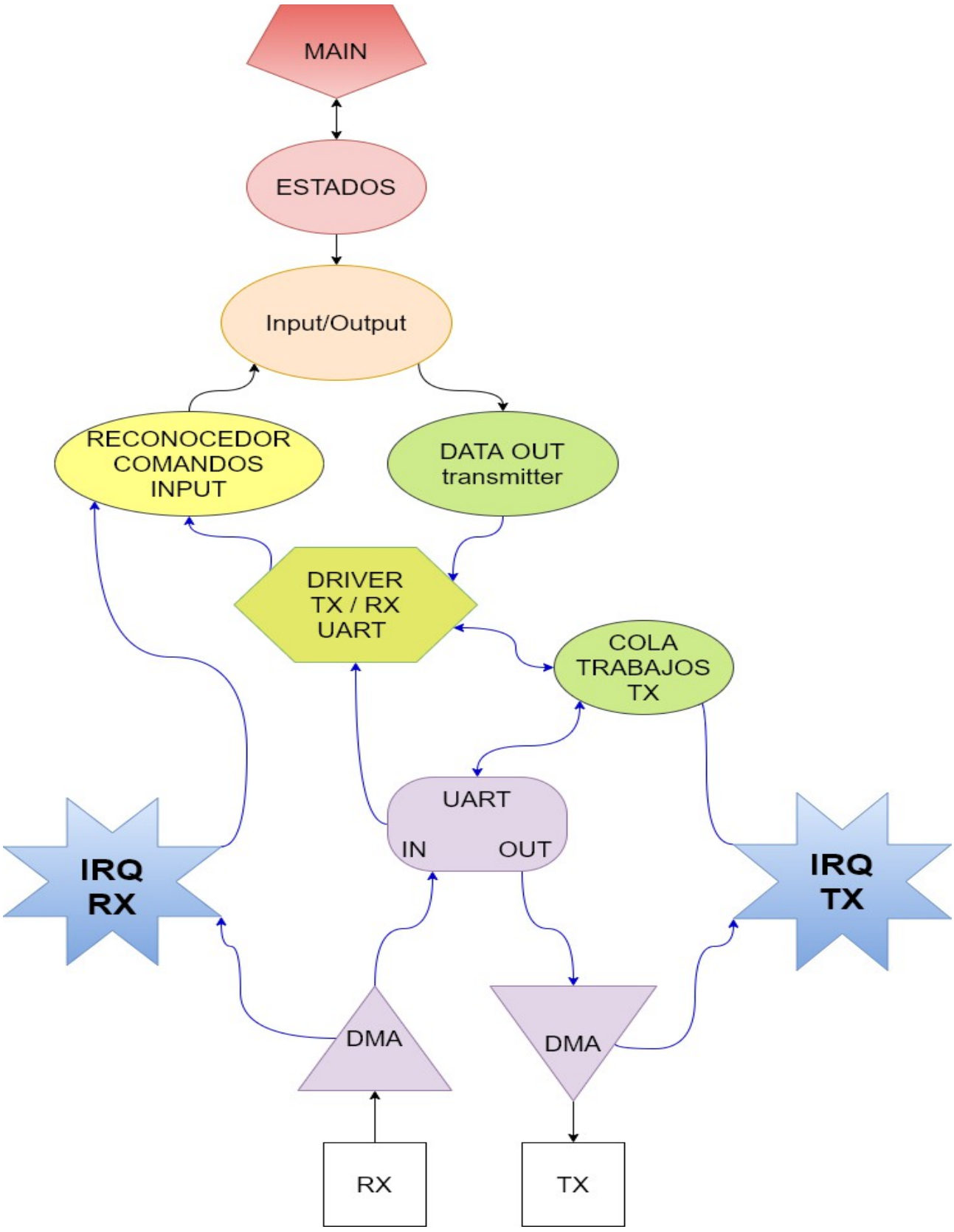
En este driver se ha unificado la parte input con la parte output, porque no era viable tener el acceso a un único recurso separado.

Si que se ha mantenido separada la parte del útil, comandos IN y transmission OUT, porque realmente se ocupa de cosas distintas.

Las **input** son el **reconocimiento de los comandos**. Se manda un *echo* cada vez que se recibe uno, así se puede comprobar si se ha transmitido correctamente.

Las **output** son el mensaje de **bienvenida**, los **echo** (ordenados desde el reconocedor de comandos de entrada) y las **transmisiones de estado**.

A continuación se puede ver un esquema de las relaciones entre los módulos que componen las comunicaciones:


MAIN
ESTADOS
Input/Output
RECONOCEDOR COMANDOS INPUT
DATA OUT transmitter
DRIVER TX / RX UART
COLA TRABAJOS TX
UART
IN
OUT
IRQ RX
IRQ TX
DMA
DMA
RX
TX

Estructura de la cola de trabajos:

Fue necesario crear una cola de **trabajos de escritura** porque las comunicaciones por UART son muy lentas en comparación con el proceso. El resto del código va más rápido y en ocasiones, los mensajes de salidas se superponían.

Una solución de compromiso fue poner un retardo entre los mensajes, pero era una chapuza.

Hubo muchas dudas a la hora de encajar este tipo de dato, pero a medida que se iba implementando se iban teniendo más claras las dependencias. Este tipo de dato pertenece al driver. El error fue pensar que por tener estructura de útil tendría que estar en un nivel superior, pero no. De ahora en adelante, se debe tener en cuenta que estos datos, con un proceso a tan bajo nivel, tienen que depender del driver.

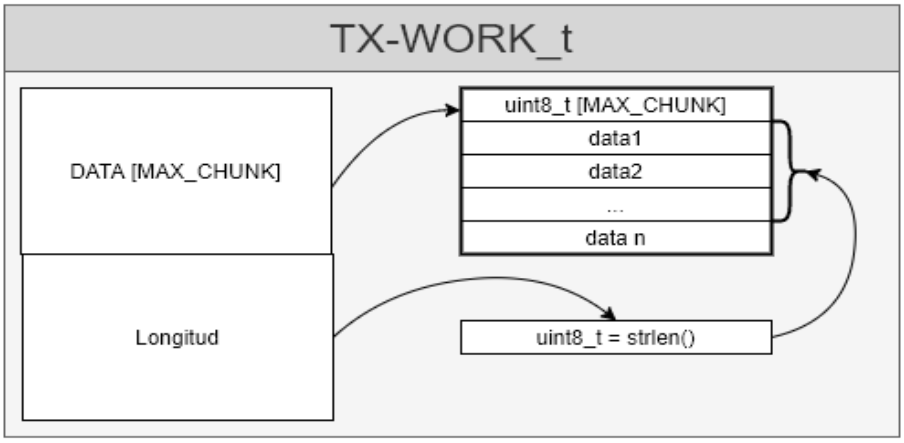
Tipos de datos:

Trabajo de transmisión

Existen dos tipos de dato; uno es la unidad de trabajo de transmisión. Estos datos que se transmiten no tienen que ser necesariametne cadenas de texto, pueden ser enteros, float... etc.

En caso de que no sean cadenas de texto, no se puede usar strlen() para conocer su longitud, se tiene que almacenar aparte.

```
#define MAX_CHUNK (uint8_t)20
typedef struct TX_Work_Struct {
        uint8_t data[MAX_CHUNK];
        uint8_t Longitud;
} TX_Work_t;
```



Clase cola

Es una **cola circular**, con operaciones **push** y **pop**. Tiene un índice para apuntar al principio (el trabajo pendiente) y un tamaño para apuntar a donde se tienen que añadir los nuevos trabajos.

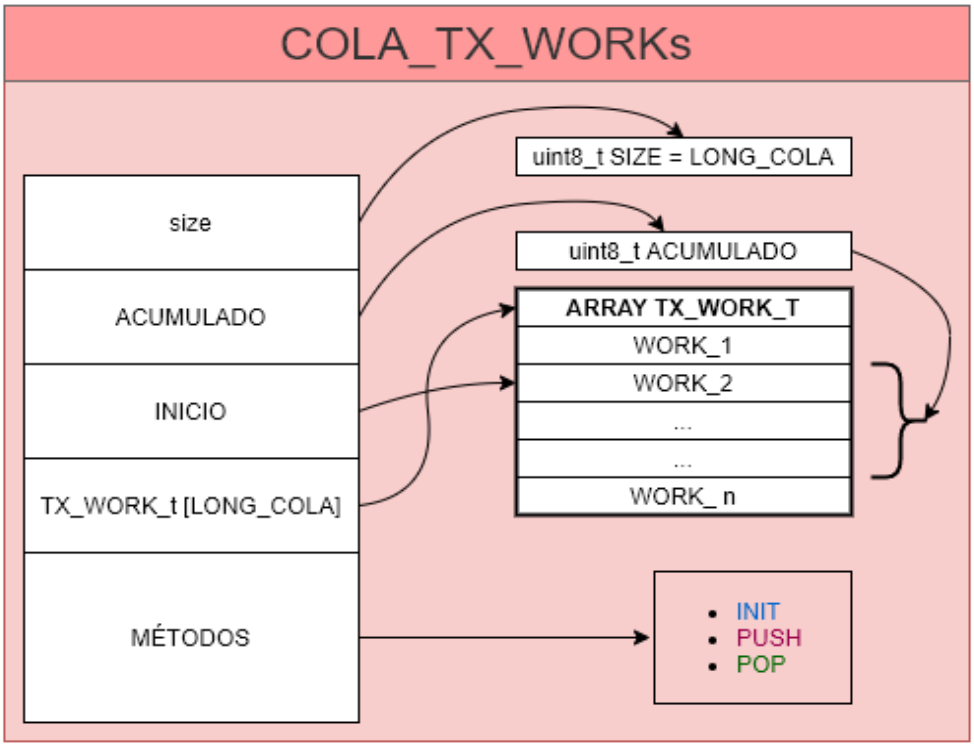
Se guarda un array con los trabajos pendientes: TX_Work_t work[UART_LONG_COLA_WRITE];

TODOS LOS TRABAJOS SE COPIAN. No se guardan por referencia unos punteros, porque eso puede suponer que se modifiquen entre que llegan y que se realiza la transmisión. Supone un consumo de memoria que en un F0 es importante.

Se guarda una variable size para tener una información del tamaño admisible.

```
#define UART_LONG_COLA_WRITE (uint8_t)20
struct StructColaUart;
typedef bool (* cola_UART_operation_push_t)( struct StructColaUart * cola, TX_Work_t * Work);
typedef bool (* cola_UART_operation_t)( struct StructColaUart * cola);

typedef struct StructColaUart {
	//datos
	uint8_t size;	// = UART_LONG_COLA_WRITE
	uint8_t acumulado; // cantidad de trabajos pendientes que contiene
	uint8_t inicio;	// puntero al inicio de la cola
	TX_Work_t work[UART_LONG_COLA_WRITE];
	//métodos:
	cola_UART_operation_push_t push;
	cola_UART_operation_t pop;
	cola_UART_operation_t init;
} ColaUART_t;
```



Métodos de la cola:

Existe un constructor, aparte de los métodos de la popia cola. Sólo crea la variable, no la inicializa.

Los métodos son:

Init:

Pone todos los datos a cero. Las cadenas del vector de trabajos, el índice y el nº de trabajos acumulados.

Pop:

Primero se comprueba que la cola no este vacía, consultando "acumulado".

Si no lo está, elimina el último trabajo, símplemente desplazando una posición el inicio de la cola.

Se comprobó que si no se borraba el trabajo daba lo mismo.

Push:

Primero se comprueba que la cola tiene al menos una posición libre.

Luego, añade un trabajo, copiando el trabajo de entrada en el vector de trabajos, primero el texto (hasta lo que indique *Longitud*) y luego el propio entero que determina la longitud.

Ese *TX_Work_t* de entrada de la función SI es temporal, se puede destruir nada más retornar de la función.

Se incrementa el "*acumulado"* una posición, usando una operación en módulo, con el operador %, para no salirse del tamaño de la cola.

Arbitro de cola

Aqui se llega desde la INT_IRQ del DMA y cada vez que se hace un PUSH de un trabajo

La función de este árbitro es examinar la cola para ver si hay otro trabajo pendiente

- Si ? -> se manda a escribir
- No ? -> se manda un trabajo de lectura indefinido; sólo parará cuando entre otro trabajo de salida

```
bool ArbitroCola (struct TXRXStruct * TXRX){
	if (TXRX->status == false){return false;} // hay algún trabajo enviándose, no lo interrumpimos

	if (TXRX->cola->acumulado >0){ // Hay trabajos de transmisión pendientes
		TXRX->status = false;
		// Se asignan los datos del trabajo que toca transmitor a las variables internas de la clase RXTX
		TXRX->String = (char*)&TXRX->cola->work[TXRX->cola->inicio].data;
		TXRX->longitud = TXRX->cola->work[TXRX->cola->inicio].Longitud;
		TXRX->write(TXRX);
	}
	else{ // no hay trabajos de transmisión, empezamos a leer
		TXRX->status = true;
		TXRX->read(TXRX);
	}
return true;
}
```

Driver TXRX

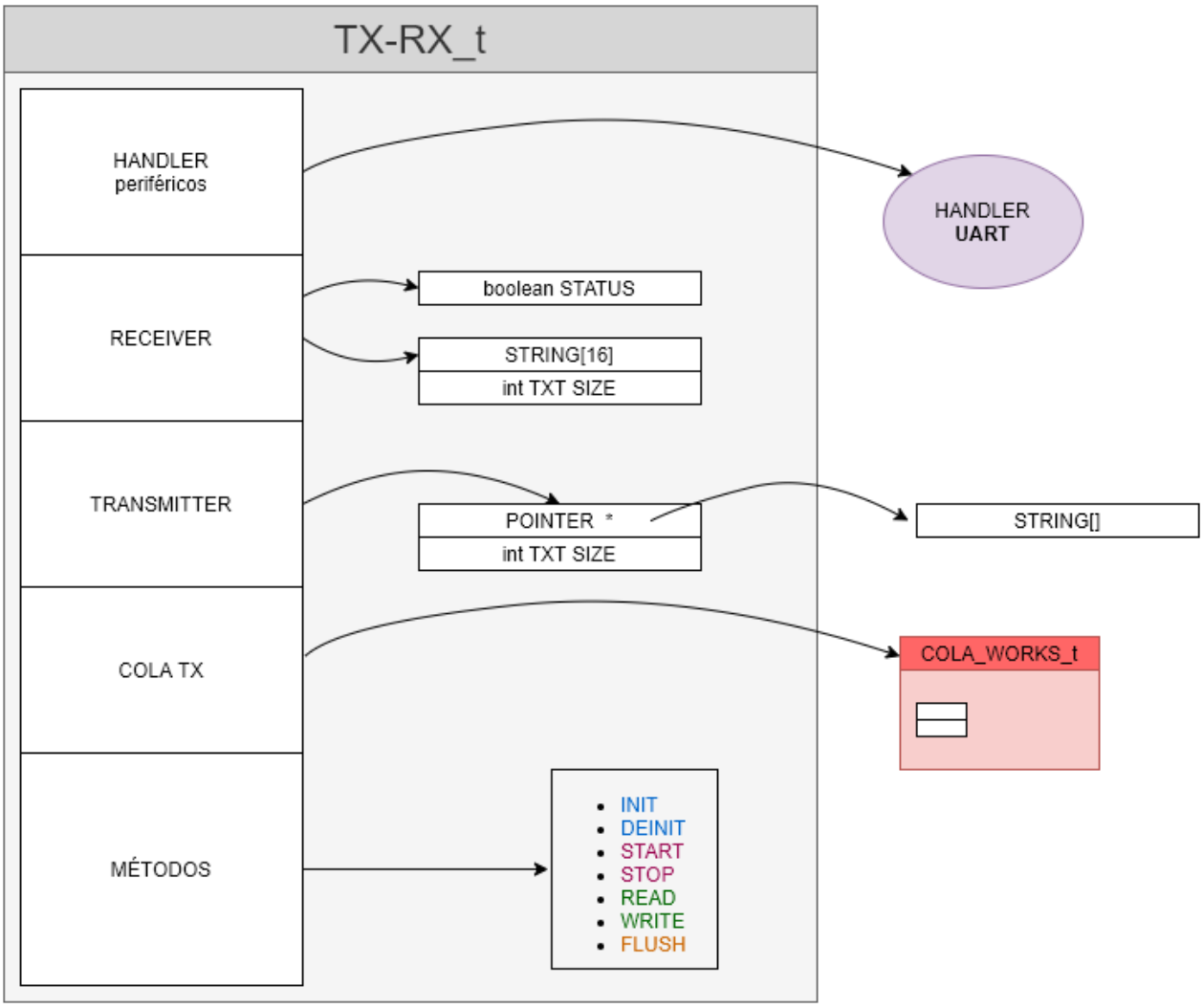
Clase del driver:

Tiene datos para la recepción, datos para la transmisión y como métodos: tiene init, start/stop, read/write, y flush. Deinit no está programado

Datos:

```
typedef struct TXRXStruct {
// datos
        // Handlers:
        UART_HandleTypeDef * uart;
        //Parte Receiver: comandos de entrada
        bool status;
        char InComTxt[LONG_IN_COM_TXT];
        uint16_t TXTsize;
        //Parte Transmitter: Echos e info de salida
        char * String;
        uint16_t longitud;
        //Parte Transmitter: Cola de trabajos de salida:
        ColaUART_t * cola;
        //métodos...
```

- El handler apunta al driver HAL que maneja la UART.
- Status es un flag que indica si la cola está transmitiendo algún trabajo de transmisión (TX_WORK_t). Ese flag lo maneja el árbitro de la cola.
- InComTxt es la cadena de texto recibida por DMA (canal RX). Se recibe sin más intervención y cuando se llena, salta la interrupción del DMA-RX (que se gestiona en el útil COMANDO IN).
- TXTsize es un entero para poder disponer del *define* LONG_IN_COM_TXT



Init:

En el **init**, se crea la cola de trabajos, y se inicializa.

```
//Inicialización de la cola de trabajos de transmisión
static ColaUART_t *colaTXRX;
colaTXRX = NewColaUART();
colaTXRX->init(colaTXRX);
TXRX->cola = colaTXRX;
```

Start y stop:

En estas dos funciones se inicia el trabajo. Sólo se para el DMA en la escritura.

Pero si que es importante señalar que se inhibe la interrupción a mitad de trabajo:

Read y write:

En estas dos funciones se inicia el trabajo. Sólo se para el DMA en la escritura.

Pero si que es importante señalar que se inhibe la interrupción a mitad de trabajo:

read:

```
//inhibimos el flag de Half Transfet interrupt - RX=ch3
TXRX->uart->hdmarx->Instance->CCR &= ~DMA_CCR_HTIE;
```

write:

```
//inhibimos el flag de Half Transfet interrupt - TX=ch2
TXRX->uart->hdmatx->Instance->CCR &= ~DMA_CCR_HTIE;
```

Lo que si tiene de especial es el árbitro. Se encarga de gestionar el tráfico de la UART, para evitar que se pisen los trabajos y minimizar el tiempo en el que no se puede recibir.

Util commando IN

Rutina de servicio de interrupción del DMA – RX

Aqui el sistema no tiene que hacer nada hasta que el DMA avise de que ha recibido los 8 caracteres que componen un comando, para esto genera una interrupción, y durante el servicio a esa interrupción, se examina la secuencia y si reconoce un comando válido, se activa su flag correspondiente y borra la cadena de entrada (InComTxt - String[16]) para que no de otros conflictos.

Como las rutinas de servicio de interrupción no admiten parámetros, se usa una variable global, ***RXCommInt***, que se declara en el driver TXRX. Esta variable también la usa el módulo TX para su rutina de servicio de interrupción del DMA-TX.

Debería ser del tipo *driver*, TXRX_t, pero hubo un error que ya está resuelto. En el módulo IO's, se asignaba en el constructor del array de IO's, justo después del objeto IO *COMMANDO_IN*. Este objeto contiene el útil *UtilComm_IN_t*, que a su vez contiene el driver *TXRX_t*, el problema residía en que cuando se asignaba, el driver aún no estaba creado, porque no se crea en el momento en el que se crea el útil, sino cuando se inicializa.

Por eso, al ejecutar la rutina de servicio de interrupción, daba un ***hard fault***, porque en el momento de la inicialización, estaba apuntando a un lugar indeterminado.

Esto es error de la arquitectura, e implica que se debe encontrar otro punto en el que asignar las variables que se emplean en las rutinas de servicio de interrupción

** Y ese momento podría ser en la inicialización de los drivers, que es justo cuando se crea el propio driver.*

Código de la rutina de interrupción:

```
void DMA1_Channel5_IRQHandler(void){
        if (Analizador_Com_IN (RXCommInt)) {// Ha reconocido un comando válido
                // Ahora la borramos para que no cree conflictos
                for ( int i=0 ; i < RXCommInt->RXcomm->TXTsize ; i++){
                        RXCommInt->RXcomm->InComTxt[i] = '\0' ;
                }
        }
  HAL_DMA_IRQHandler(&hdma_usart1_rx);
}
```

Reconocedor de comandos:

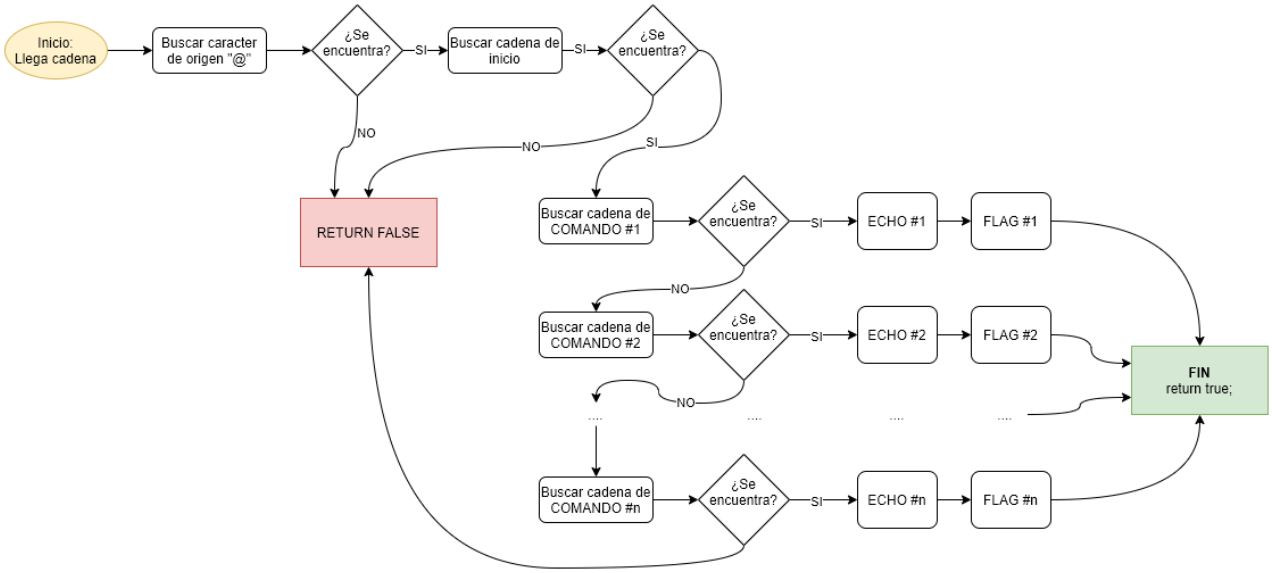
Func

```
bool Analizador_Com_IN (TXRX_t * Rxcomm);
```

Es un reconocedor de secuencias muy básico, en primero lugar busca el caracter de inicio. Al emplear un DMA con una cola circular, nos puede entrar ruido y el principio de la cadena estará en una posición indeterminada.

```
        if (inicio == RXcomm->size){// Comm initial character not found
```

SI no lo encuentra, se termina el algoritmo, si SI, sigue y se empieza a buscar a partir de esa posición.



Luego busca la cadena "PCP", si la encuentra sigue. Luego busca secuencialmente las cadenas de los distintos comandos.

Cuando encuentra una válida, genera un ECHO y activa el flag que hay en la estructura de datos del propio driver, que tiene unos bool como flags para los comandos de entrada.

Cuando se leen, se borran, eso ya en otros módulos.

Util Transmitter

tiene tres funciones públicas, se pueden ampliar en un futuro:

void **trasmitHello** (TXRX_t* Trans);

void **trasmit_Status** (TXRX_t* Trans, Estados_t Estao, bool EsMuted);

void **Echo** (TXRX_t * RXcomm, char* echo);

Son muy obvias.

En un futuro se añadirá algo que permita transmitir tablas de los valores que capta el ADC, directamente en ASCII para que se puedan interpretar con excel como un fichero CSV.

La importante es que la escritura (función privada) se hace desde este útil.

trasmitWrite

bool **trasmitWrite**(TXRX_t* Trans, char * mensaje);

Esta función genera un trabajo de transmisión: TX_Work_t, con el mensaje que le llega.

Comprueba si es demasiado grande, más que el tamaño máximo de cada trabajo, multiplicado por el número de trabajos que caben en el array de la cola.

Si es demasiado grande, devuleve false.

Si no, comprueba si cabe todo dentro de un trabajo, y si no, lo trocea – **SLICER**.

Esta subrutina, no declarada, es chorras pero muy importante. Permite que no se atragante, y permitirá introducir lecturas intercaladas, o retrasos de cualquier tipo en la UART.

```
    TX_Work_t worky;
    uint32_t longmensaje;
    longmensaje = strlen(mensaje);

    if ( longmensaje > MAX_CHUNK*UART_LONG_COLA_WRITE){return false;}
    // AQUI VA EL SLICER
        uint32_t index=0;
        while ( longmensaje >= MAX_CHUNK){
            for (int i=0; i< MAX_CHUNK; i++){
                worky.data[i]= (uint8_t)*(mensaje + index);
                index++;
            }
            worky.Longitud = MAX_CHUNK;
            Trans->cola->push(Trans->cola, &worky);
            longmensaje -= MAX_CHUNK;
        }
        // Ahora que el mensaje ya es más pequeño que el CHUNK se mete todo entero
        strcpy ((char*)worky.data, (mensaje + index));
        worky.Longitud = longmensaje;
        Trans->cola->push(Trans->cola, &worky);
    return ArbitroCola(Trans);
}
```

Rutina de servicio de interrupción del DMA - TX

El DMA es quien genera la interrupción cuando se han completado tanto las transmisiones como las recepciones.

Todo funciona por esto. Es sumamente importante.

Primero discrimina si la interrupción la crea la recepción o la transmisión

- Si es recepción, manda a reconocer la cadena
- Si es transmisión, borra el último trabajo y llama al árbitro

```
void DMA1_Channel2_3_IRQHandler(void) {/* USER CODE BEGIN DMA1_Channel2_3_IRQn 0 */
        bool EsTX = false;
        if (DMA1->ISR & DMA_ISR_TCIF2){ // Transfer Complete channel 2 - TX
                EsTX = true;
        }
        if (DMA1->ISR & DMA_ISR_TCIF3){ // Transfer Complete channel 3 – RX
                if (Analizador_Com_IN (RXCommInt)) {
                        // Ha cazado una cadena válida, la borramos para que no se vuelva a reconocer
                        for (int i=0; i<RXCommInt->size; i++){
                                RXCommInt->InComTxt[i]= '\0';
                        }
                }
        }
  /* USER CODE END DMA1_Channel2_3_IRQn 0 */
  HAL_DMA_IRQHandler(&hdma_usart1_tx);
  HAL_DMA_IRQHandler(&hdma_usart1_rx);
  /* USER CODE BEGIN DMA1_Channel2_3_IRQn 1 */
        if (EsTX){
                DMA1->IFCR &= ~DMA_IFCR_CTCIF2; // bajamos el flag de transfer complete
                DMA1->IFCR &= ~DMA_IFCR_CHTIF2; // bajamos el flag de half transfer
                DMA1->IFCR &= ~DMA_IFCR_CGIF2; // bajamos el flag de global interrupt
                //Flag de status
                RXCommInt->uart->gState= HAL_UART_STATE_READY;
                RXCommInt->status = true;
```

```
            //hacemos un pop de este trabajo
            RXCommInt->cola->pop(RXCommInt->cola);
            //Y llamamos al árbitro
            ArbitroCola(RXCommInt);
        }
  /* USER CODE END DMA1_Channel2_3_IRQn 1 */
}
```

Listado de comandos

Existen 4 comandos, están listados en el archivo "Command2.c"

Estructura del archivo:

La estructura de este archivo es

- Defines de número de comandos y longitud (NUM_COMMANDS_UART entre otros)
- Defines / Etiquetas para indexar los vectores
- bool BoolCommandList[NUM_COMMANDS_UART]
  - *Esta es la lista de flags asociada a los comandos*
- char * CharCommandList[NUM_COMMANDS_UART]
  - *Esta es la lista de comandos*
- char * EchoCommandList[NUM_COMMANDS_UART]
  - *Esta es la lista de textos de **echoes** para cada comando*

Lista de comandos:

La lista de comandos se compone de 4 comandos para cambiar de canal. No hay "stat" ni "id"

Es la siguiente:

"CH1_",
"CH2_",
"CH3_",
"CH4_"

Finalmente, se reconocerá el texto inicial con el caracter de inicio: "@" + "PCP", y quedará un comando de este tipo:

@PCPCH1_

Versión "finalizable"

En esta versión "finalizable" se deberían hacer los siguientes cambios:

- Eliminar las funciones de test del *main*.
- Incluir y configurar el watchdog.

Control de versiones

versión	Cambios	fecha
'0.0	Versión inicial del documento-("Plantilla de documento.odt")	01/09/19
'1.0	Creación del documento de diseño de Software.	09/03/20
'1,0	Finalización del documento de diseño de software	25/06/20
1,0,1	Incuye los últimos cambios para generar un software finalizable	06/07/20
'1.0.2	Se incluye la lista de comandos y la descripción de entradas y saliads físicas, así como el pin-out del microcontrolador	29/07/20