Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Desarrollo de aplicaciones con el entorno TouchGFX para sistemas embebidos

Autor: Jorge Celso Camacho

Tutor: José Manuel Villadangos Carrizo

2017

UNIVERSIDAD DE ALCALÁ
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Electrónica de Comunicaciones

Trabajo Fin de Grado

Desarrollo de aplicaciones con el entorno TouchGFX para sistemas embebidos

Autor: Jorge Celso Camacho
Tutor: José Manuel Villadangos Carrizo

TRIBUNAL:

Presidente: Julio Pastor Mendoza

Vocal 1º: Juan Manuel Miguel Jiménez

Vocal 2º: José Manuel Villadangos Carrizo

FECHA**:** Septiembre 2017

Resumen

El interfaz entre el hombre y la máquina —IHM— es un elemento imprescindible en todo sistema informático o electrónico con el que poder interactuar. El entorno gráfico de usuario —GUI— es una variante del anterior, cuya pretensión es la de facilitar esta interacción mediante el uso de controles gráficos, presentando en ellos información o facilitando el control. Inicialmente, los entornos gráficos se implantaron en los ordenadores de sobremesa para luego pasar a los teléfonos móviles. Esta tendencia se ha establecido también en los sistemas embebidos, donde el GUI se ha convertido en una pieza a tener en cuenta a la hora de realizar un desarrollar.

El presente trabajo estudia la estructura e implementación de los entornos gráficos de usuario en sistemas embebidos con recursos limitados[1], centrado en el paquete software «*TouchGFX*».

El objetivo es explicar el funcionamiento interno de esta librería, poniendo especial interés en la comunicación entre el usuario y el entorno gráfico, entre los distintos elementos que constituyen el entorno, y especialmente, entre este y la aplicación final. Para ello, se crea una aplicación que presente, gráficamente, la Transformada Rápida de Fourier de una señal adquirida.

Palabras clave: Entornos gráficos de usuario, sistemas empotrados, TouchGFX, sistema operativo en tiempo real, programación orientada a objetos.

[1] En comparación con los sistemas empotrados constituidos por los teléfonos móviles actuales.

Abstract

The interface between human and machine -IHM- is an essential element in any computer or electronic system to interact with. The graphical user interface -GUI- is a variant of the previous one, whose intention is to allow this interaction through the use of graphic controls, showing information in them or allow the system control. Initially, graphical environments were deployed on desktops and then switched to mobile phones. This trend has also been established in embedded systems, where the GUI has become a piece to take into account when developing an embedded system.

The present work studies the structure and implementation of user graphical interface in embedded system with limited resources, focused in the software package "*TouchGFX*".

The target is to explain the internal operation of this library, putting special interest in the communication between the user and the graphic environment, between the different elements that make up the environment, and especialy, between this one and the final application. To do this, an application is created that plots the Fast Fourier Transform of an acquired temporal signal.

Índice general

Índice de figuras

Índice de Tablas

Índice de Listado de Código Fuente

Listado de Acrónimos

ADC	Analog to Digital Converter
AF	Alternative Function
AHB	Advanced High-performance Bus
APB	Advanced Peripheral Bus
API	Application Programming Interface
ARM	Advanced RISC Machine
CLK	Clock
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit
DAC	Digital to Analog Converter
DIP	Dual Inline Package
DMA	Direct Memory Access
DMA2D	Direct memory Access two dimensions
DMIPS	Dhrystone Millions of Instructions Per Second
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First In, First Out
FILO	First-In, Last-Out
FMC	Flexible Memory Controller
FPC	Flexible Printed Circuit
FPU	Floating Point Unit
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HBP	Horizontal Back Porch
HFP	Horizontal Front Porch
HIM	Human Interface Machine
HSE	High Speed External clock signal
HSL	Hue, Saturation, Lightness
HSV	Hue, Saturation, Value
I2C	Inter-Integrated Circuit
IOE	Input/Output Expander
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
LCD	Liquid Cristal Display
LIFO	Last In, First Out
LSB	Least Significant Bit
LTDC	LCD-TFT controller
MSB	Most Significant Bit
MSPS	Megasamples Per Second
NVIC	Nested Vectored Interrupt Controller
OSC	Oscilator
PCB	Printed Circuit Board
PLL	Phase Locked Loop
RC	Resistencia-Condensador
RCC	Reset and Clock Control
RGB	Red, Green, Blue
RTOS	Real-Time Operating System
SAI	Serial Audio Interface
SPI	Serial Peripheral Interface
TFT-LCD	Thin Film Transistor-Liquid Crystal Display
TSC	Touch Screen Controller
VBP	Vertical Back Porch
VFP	Vertical Front Porch
XGA	Extended Graphics Array 1024 x 768

Introducción

Este trabajo se divide en cuatro capítulos: el primero dedicado a presentar la librería utilizada, *TouchGFX*; el segundo, a mostrar el sistema operativo en tiempo real *FreeRTOS*, que viene integrado en la librería gráfica; el tercero, destinado a explicar la realización física del proyecto y el cuarto, a exponer la aplicación software con entrono gráfico, que realmente representa la elaboración del proyecto.

En el primer capítulo se presenta la descripción y forma de utilización de la librería TouchGFX. Inicialmente se muestran conceptos generales, como son las posibles topologías de un sistema empotrado o los dispositivos, externos o internos, involucrados en el tratamiento de datos gráficos. A continuación se pasa a describir el paquete software que el fabricante suministra. Aquí están incluidos tanto las posibles distribuciones como el hardware soportado, la estructura de directorios de la librería y la estructura de directorios que ha de tener una nueva aplicación. También se muestran los pasos a seguir para crear una nueva aplicación TouchGFX. Luego se pasa a exponer conceptos teóricos como son las temporizaciones que utiliza el entorno, la estructura de capas de la librería o el patrón de programación implementado. Posteriormente se presenta el mecanismo de comunicación de los elementos gráficos con el entorno, la creación de nuevos componentes y la descripción de los componentes integrados dentro de la librería. Para finalizar, se muestran dos clases, que por su especial utilidad, son usadas de forma recurrente.

En el segundo capítulo se comienza describiendo qué es un sistema operativo en tiempo real y cómo se utiliza FreeRTOS. Luego se pasa a exponer las distintas configuraciones practicadas en FreeRTOS —directivas de configuración y manejo de prioridades en interrupciones, entre otras— para funcionar con procesadores ARM y con TouchGFX, y de qué manera se realiza el anclaje de funciones a tareas para que TouchGFX pueda medir la actividad del micro. Finalmente se expone cómo se implementa la capa de abstracción del sistema operativo de TouchGFX con FreeRTOS o sin sistema operativo.

El tercer capítulo presenta la placa de desarrollo utilizada para llevar a cabo la aplicación, describiendo sus características y las del microcontrolador que incluye. Luego, se pasa a describir el display utilizado, exponiendo sus características y mostrando el diseño de la placa que lo conecta a la placa de desarrollo, así como las modificaciones practicadas en esta. La parte final del capítulo trata la configuración de TouchGFX respecto al software y hardware para que refleje el cambio de display, poniendo especial atención al nuevo driver táctil desarrollado.

El cuarto capítulo representa el desarrollo del proyecto de entorno gráfico de usuario en sí. Inicialmente se expone el diseño gráfico de pantalla de la aplicación, mostrando los distintos menús y controles necesarios. Luego, el resto del capítulo se divide en cuatro partes. En la primera, se tratan aspectos ajenos al entrono gráfico, como es la modificación de la frecuencia de reloj del sistema y la configuración de diversos periféricos internos del microcontrolador para generar varias señales de prueba. En la segunda, se muestran los controles gráficos utilizados que van a necesitar desarrollo, ya sea partiendo desde cero, o desde controles que el fabricante utiliza en ejemplos. En la tercera, se presenta la aplicación final de usuario, es decir, la aplicación que realmente realiza una tarea útil, y no pertenece al entorno gráfico, pero que debe ser controlada desde este y presentar su información. En la parte final se expone el desarrollo del entorno gráfico, mostrando las modificaciones practicadas a su patrón de programación y la comunicación entre sus distintos elementos y con la aplicación final.

Vista la estructura de capítulos, se puede asignar un carácter más teórico a los dos primeros y más práctico a los dos segundos. Sin embargo, esta distinción no es tan estricta. Así, por ejemplo, munchos de los códigos ejemplo presentados en los dos primeros, están cogidos directamente del código del proyecto, o con leves

modificaciones. Por otro lado, en los dos últimos capítulos se presentan conceptos teóricos como son el funcionamiento de una pantalla táctil resistiva de cuatro hilos —capítulo 3— o el algoritmo FFT de «diezmado en frecuencia»—capítulo 4—.

Tanto el modelo de capas de la librería TouchGFX como el patrón de programación que implementa, utilizan la palabra «aplicación» —application— para designar a alguno de sus elementos. Sin embargo, en este proyecto se denota con «aplicación» a la aplicación que realiza la tarea útil para el usuario y que no perteneciente al entrono gráfico.

1 Descripción del entorno gráfico TouchGFX

1.1 Introducción

Ya que el proyecto trata de la creación de un interfaz gráfico de usuario —GUI—, a través de la herramienta comercial *TouchGFX*, y debido a las posibilidades topológicas que tiene un sistema empotrado con capacidad gráfica, se hace necesario conocerlas para poder seleccionar el hardware más conveniente, teniendo en cuenta las necesidades de este entrono comercial así como la posibilidad de realizar modificaciones sobre el hardware soportado. Así mismo, siempre se ha de valorar criterios económicos.

1.1.1 Modelos hardware para la gestión del display

Existen cuatro elementos fundamentales que intervienen en un sistema gráfico que tenga cierto grado de complejidad de todo dispositivo empotrado, que son: el microcontrolador, la memoria gráfica, el controlador de pantalla y la pantalla en sí misma. Cada uno de estos elementos está representado en la siguiente figura [30]:

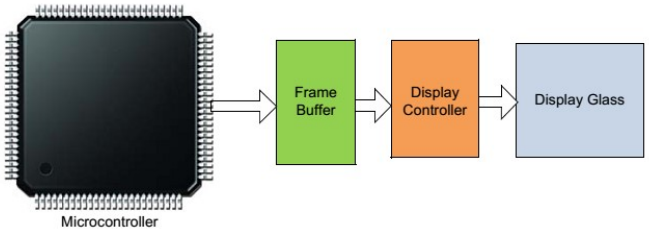


Ilustración 1.1: Elementos constituyentes de un sistema gráfico

El microcontrolador es el encargado de mandar la información a la memoria gráfica, generalmente refrescando su contenido coincidiendo con la señal de sincronismo vertical de pantalla —que genera el controlador de pantalla—. Este envío de información puede proceder de algún otro periférico interno o externo, donde esté albergada la información gráfica. En este caso, habitualmente se trata de un tipo de memoria ROM, y el trasvase de información desde esta hacia la memoria gráfica se suele realizar mediante acceso directo a memoria —DMA—. El DMA, frecuentemente, es un periférico interno del microcontrolador, y su funcionamiento

puede ser genérico, moviendo bloques de memoria de un sitio a otro, o especializado para cuestiones gráficas, en cuyo caso estos bloques de memoria tienen representación bidimensional, correspondiendo a regiones rectangulares de pantalla —como es el caso del periférico DMA2D de la serie de micros STM32 del fabricante ST, utilizado en este proyecto—. En el caso de no disponer de este periférico, ha de ser el micro el que se deba encargar de esta tarea, con el consiguiente consumo de tiempo y la pérdida de rapidez para la ejecución del programa. Este tipo de información gráfica es denominada mapa de bits. Por otro lado, la fuente de datos gráfica puede ser el propio micro que los genere y modifique en tiempo de ejecución. En este caso se habla de gráficos vectoriales y siempre consumen tiempo de ejecución. En la medida de lo posible, este segundo tipo de gráficos ha de evitarse, aunque existen aplicaciones en las que no es viable, como pueda ser la representación de datos, actualizados de forma permanente, en coordenadas cartesianas, o figuras geométrica en transformación —desplazamiento, rotación o ampliación—, como la aguja de un medidor de tipo «reloj».

El buffer de memoria ha de ser de tipo RAM y de acceso lo más rápido posible. En él, el microcontrolador almacenará la información gráfica que ha de ser mostrada en pantalla, refrescándola de forma periódica para poder actualizar cambios que sucedan en ella. Este refresco ha de estar sincronizado con el barrido que efectúa el controlador para mandar la información a la pantalla y ha de hacerse de forma que no produzca efectos indeseados en la representación gráfica —artefactos—. Se suele utilizar el tiempo de borrado vertical de pantalla —momento en el que no se está mandando información visible a pantalla y sucede al dibujar un nuevo cuadro— para actualizar la información en el buffer. Otra técnica empleada, en conjunción con la anterior, es utilizar el doble buffer, que consiste en dividir el buffer en dos partes de igual tamaño entre sí, e igual al tamaño necesario para poder albergar la información que cubra toda la pantalla. Así, mientras el controlador usa uno de ellos para refrescar la pantalla de forma permanente —buffer activo—, el microcontrolador usa el otro —buffer secundario— para elaborar la representación gráfica que se usará a continuación. En el borrado vertical, en vez de actualizar toda la representación gráfica, lo único que se hace es intercambiar los buffers entre sí. Esto no es más que intercambiar las direcciones de memoria de ambos, algo que sucede de forma muy rápida. Este último modo de funcionamiento es el que emplea *TouchGFX*, aunque puede ser configurado para emplear un buffer único.

El controlador de pantalla es el encargado de barrer el buffer gráfico en la memoria RAM y mandar la información allí codificada hacia la pantalla, en el formato de puntos —información RGB—, que sea comprensible por esta. Para ello, genera una serie de señales de temporización o sincronización que necesita la pantalla para ir mostrando cada punto que le es enviado. Estas señales son el sincronismo horizontal —que indica a la pantalla cuando iniciar el dibujado de una nueva línea—, el sincronismo vertical —que indica a la pantalla cuando iniciar la representación de un nuevo cuadro— y el reloj de puntos, que indica a la pantalla cuando mostrar el punto que tiene en su bus de datos. Este bus de datos es por donde el controlador manda la información de cada punto en formato RGB —rojo, verde, azul—, y dependiendo del par controlador-pantalla, puede ser analógico —tres líneas de datos, una para cada color—, o digital —varias líneas por cada color—. Existen pantallas que necesitan una señal de control más, denominada habilitación de datos, que permanece activa —o desactiva, según la pantalla utilizada— durante la transferencia de puntos en cada línea. En algunos casos, esta señal puede sustituir las de sincronización horizontal y vertical —como es el caso de este proyecto—.

Dependiendo donde se localice cada uno de estos elementos, se dispone de las siguientes topologías [30]**:**

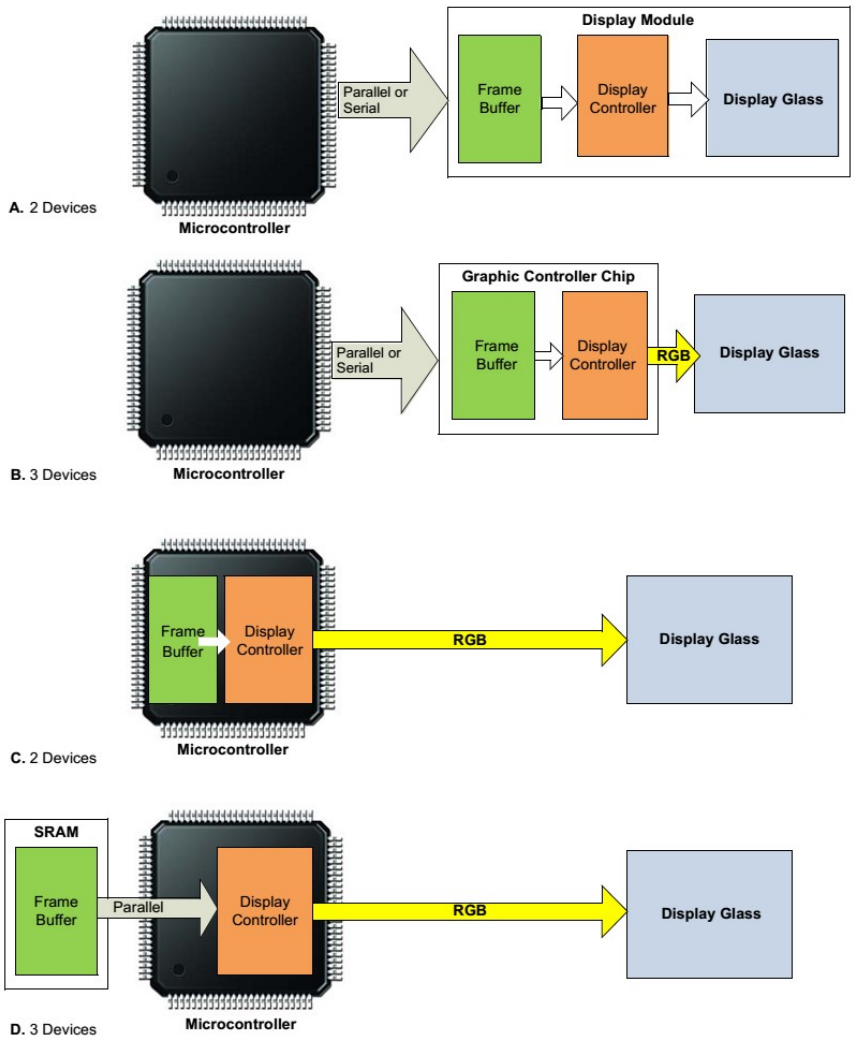


Ilustración 1.2: Topologías posibles de los elementos en un sistema gráfico

En el primer caso mostrado tanto el buffer gráfico como el controlador están alojados dentro de la pantalla. El microcontrolador se comunica con el módulo de pantalla a través de un bus paralelo o serie. La principal ventaja, es que descarga de responsabilidad al micro respecto del barrido de la pantalla y el manejo del buffer gráfico. La desventaja es la pérdida de control en la representación y la poca rapidez de actualización de nueva información. Se suele utilizar en sistemas de baja potencia computacional.

En el segundo caso, el buffer y el controlador están integrados dentro de un mismo circuito. Desde el punto de vista del micro, esta topología es idéntica a la anterior. Al tener la pantalla el controlador fuera de sí, se necesitan líneas de comunicación entre ambos —RGB y señales de sincronización— con el consiguiente consumo de espacio en el circuito impreso. La principal ventaja es el abaratamiento de la pantalla y menor coste en el caso de tener que sustituirla por motivo de avería.

En la tercera topología, tanto el buffer como el controlador están dentro del micro. Esto hace que solo el bus RGB —con las señales de sincronización— se deba utilizar. La gran ventaja es la rapidez de acceso al buffer gráfico, ya que está integrado dentro del propio micro, pero la capacidad de memoria será menor que en el caso de usar un buffer externo. La consecuencia del uso de esta topología es la utilización de pantallas de resoluciones

bajas. Al estar todo integrado, requiere el uso de micros con mayores prestaciones con el consiguiente aumento de precio.

En la última topología, el buffer gráfico es externo al micro, mientras que el controlador permanece integrado. Este es un modelo compromiso entre los dos anteriores. Permite utilizar pantallas más baratas, ya que no es necesario que dispongan de una lógica compleja, y el micro mantiene el control de buffer gráfico. Esto permite la utilización de doble buffer, capacidades mayores de memoria —se pueden manejar resoluciones de pantalla mayores—, pero el tiempo de acceso a ella será algo más lento que el de la topología anterior, sin embargo, suficiente para la mayoría de las aplicaciones.

Esta última topología será la empleada en este proyecto, donde se utilizará la placa de desarrollo STM32F429I-DISCO, que contiene un micro con controlador de pantalla integrado —periférico interno LTDC— y memoria externa —SDRAM 64 Mbits—.

1.1.2 Esquema de flujo de datos hacia el display

En la sección anterior se ha mostrado parte del camino que tienen que recorrer los datos para llegar hasta la pantalla, pero no se especificó todo el recorrido. En este recorrido pueden intervenir otros elementos como la memoria no volátil y el acceso directo a memoria. El siguiente esquema muestra el camino completo [8]:

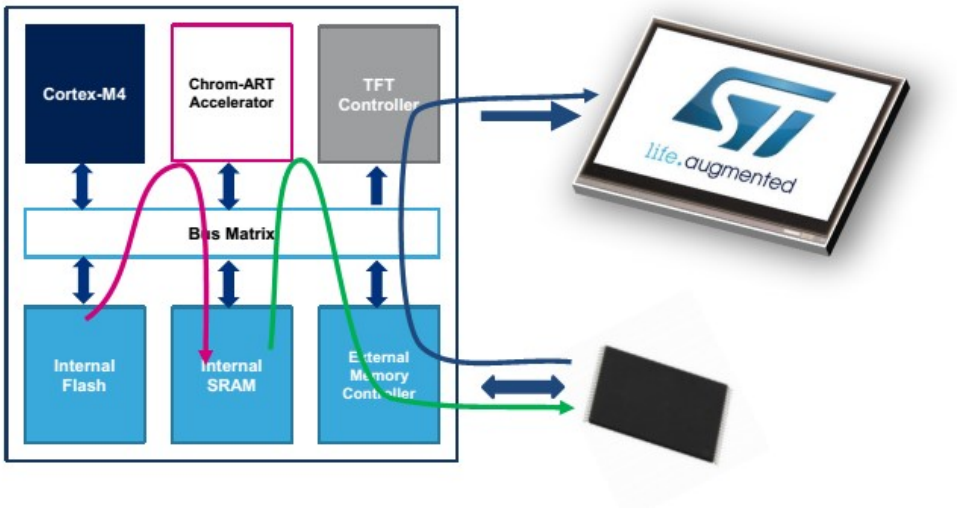


Ilustración 1.3: Camino de los datos hacia pantalla

Este es un esquema referente a microcontroladores del fabricante ST —facilitado en varios documentos como notas de aplicación o seminarios—, que poseen capacidades de control sobre una pantalla gráfica, pero la filosofía es genérica. De todos modos, el microcontrolador usado en este proyecto es de ST —STM32F429ZIT6— con las características concretas mostradas en la figura.

Normalmente, la pantalla será actualizada con información de mapas de bits que estará contenida en memoria no volátil, que puede ser externa al micro —lo que posibilita mayor capacidad—, o interna —como la representada en la figura y correspondiente con el caso particular de este proyecto— con menor capacidad. Esta información es movida por el periférico interno *DMA* a través del bus interno del micro —*Bus Matrix*— sin intervención del núcleo —*Cortex-M4*—. Concretamente, la versión específica de este periférico para datos gráficos bidimensionales, es el llamado *DMA2D* y que aparece en la figura como «*Chrom-ART Accelerator*». Este periférico puede utilizar la memoria *RAM* interna —*internal SRAM*—, a modo de caché, para realizar el trasvase hasta la memoria *RAM* externa, a través del periférico interno de control de memorias *FMC* —*External Memory*

Controller—. Por otro lado, el periférico interno del control de pantalla —*TFT Controller*—, accede a la *RAM* externa para refrescar la pantalla con la información contenida. Este último periférico, además de controlar las señales de sincronismo y datos de pantalla, posibilita el uso de dos capas sobre la misma, que pueden entenderse como «pantallas lógicas» que se mezclan, según ciertos criterios, para formar la información que es mandada a la «pantalla física». En el caso de este proyecto solo se utilizará una de las capas, con lo que la «pantalla lógica» y la «pantalla física» serán equivalentes.

Como se va a utilizar el entorno gráfico *TouchGFX*, que soporta el micro y placa utilizados en este proyecto —STM32F429ZIT6 y STM32F429I-DISCO respectivamente—, todo este camino es transparente para el desarrollador. Sin embargo *TouchGFX* necesita configurar todos estos periféricos internos para que funcionen de acuerdo a los periféricos externos que posee la placa de desarrollo. Todo esto ya está configurado dentro del entrono *TouchGFX*, sin embargo, ya que se va a modificar la placa utiliza —cambiando el display que viene instalado por otro mayor, con más resolución, y la pantalla táctil— será necesario modificar la configuración de varios de estos periféricos para acomodarse a la nueva situación. Todo ello será tratado en el capítulo 3.

1.2 Librería TouchGFX, características generales.

1.2.1 Modelo de capas en TouchGFX

TouchGFX más que una librería gráfica, es un entorno gráfico estructurado en capas —aunque en posteriores referencias a él, dentro de este documento se le denomine librería–. Está escrito en lenguaje C++ que permite realizar interfaces de usuario, para sistemas empotrados, con apariencia similar a la de los móviles actuales pero con mucho menos recursos hardware que estos. La estructura de estas capas es la mostrada en la siguiente figura [1]:

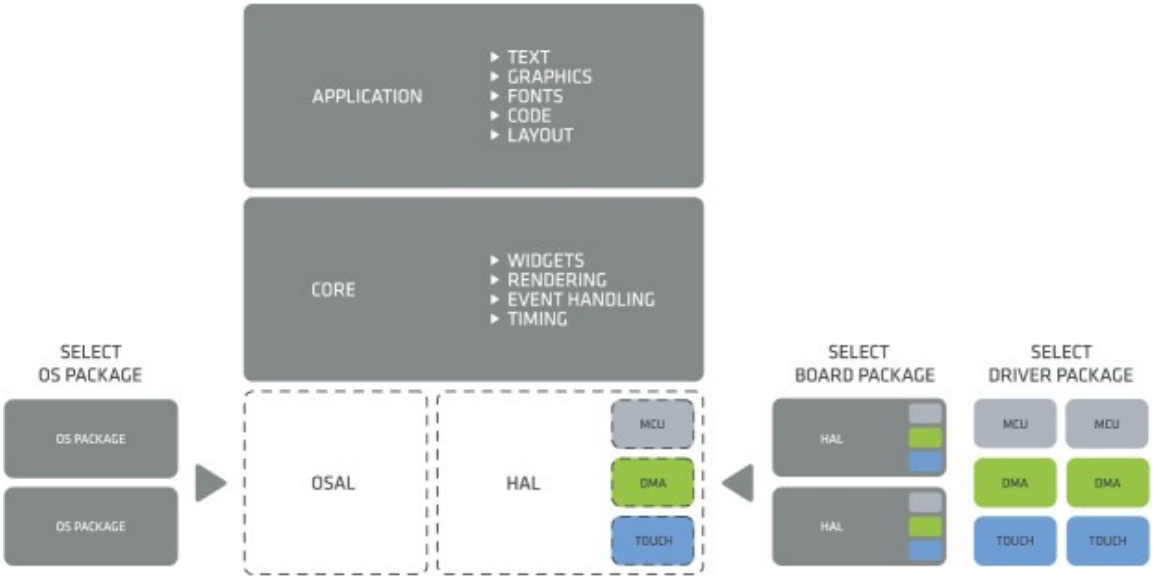


Ilustración 1.4: Modelo de componentes de TouchGFX

Consta de las capas *APPLLICATION* —aplicación—, *CORE* —núcleo—, *OSAL* —abstracción de sistema operativo— y *HAL* —abstracción de hardware—.

La capa más superior, la de aplicación, es la realmente a implementar por el desarrollador, donde especificará los elementos gráficos a utilizar en el interfaz y la lógica de comportamiento particular de los mismos. Estos elementos son las cadenas de textos utilizadas en el interfaz, la representación gráfica de cada una de ellas —las fuentes de texto—, los mapas de bits utilizados —con información de transparencia (capa alpha), o sin ella— y el código y la organización gráfica. La organización gráfica hace referencia a la gestión de contenedores —componente que contienen otros componentes—, a la estructura y conmutación entre las distintas vistas —pantallas— que componen el interfaz y a la implementación de la gestión de la comunicación entre la lógica de la aplicación, propiamente dicha, y el entrono gráfico; es lo que se conoce como patrón de programación *MVC* —modelo- vista -controlador—, o como MVP —modelo-vista -presentador—, que es la denominación que *TouchGFX* utiliza. Como elementos más importantes en esta capa —no mostrado en el esquema anterior— están las instancias u objetos de los componentes gráficos —*widgets*—, utilizados en la capa

aplicación cuyas clases están contenidas en la capa inferior. El código perteneciente a esta capa es, principalmente, el dedicado a la organización del entorno gráfico, a la instanciación de controles, a la gestión de la manipulación de los eventos de estos controles y a la comunicación entre los elementos del modelo-vista-presentación. De hecho, el código útil de la aplicación de usuario no estará contenida en esta capa ni otra en el entrono gráfico. Ya que el entrono gráfico se ejecuta en una tarea del sistema operativo utilizado —*FreeRTOS*—, sin opción a acceder al código de la tarea —ya que en la versión de evaluación de *TouchGFX*, la práctica totalidad de él está en librerías compiladas—, la única opción es colocar el código útil de la aplicación en otra tarea concurrente con la anterior —donde se localice, por ejemplo, el código de configuración y acceso a hardware interno del micro utilizado—. Por ello, en el resto de este documento, cuando se haga mención a «aplicación», se estará haciendo referencia a esta segunda tarea del sistema operativo o al código contenida en la misma.

La capa inmediatamente inferior, la capa *CORE* —núcleo, podría ser su traducción—, contiene todas las definiciones de clases de controles gráficos —*widgets*— soportados por la versión de *TouchGFX* y que pueden ser usados en la capa de la aplicación. También es la encargada la manipulación del doble buffer de pantalla —con ayuda de la capa *HAL*— y tratar la lógica de invalidación de gráficos —algoritmo *oclussion-culling* u ocultación selectiva—, realizar la gestión de los eventos, lo que implica testear si se ha producido un evento y la consecuente redirección al componente correcto, y gestionar las temporizaciones, que supone la llamada a los eventos de temporización de todos aquellos controles gráficos que hayan sido registrados para ello y la ejecución de la rutina de temporización de las vistas si estas la implementan. Esta capa, en la versión de evaluación de *TouchGFX*, está totalmente compilada en la librería touchgfx_core.lib, con lo que será necesario añadirla al entorno de desarrollo:

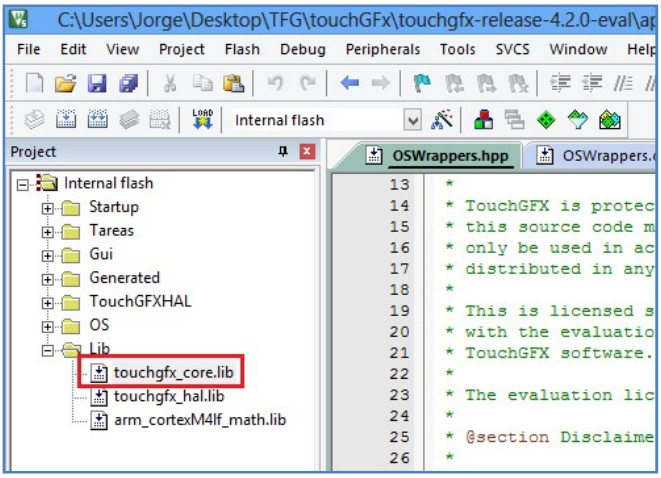


Ilustración 1.5: Inclusión de la capa CORE en el entorno de desarrollo —Keil—

En principio, esta capa no debe ser objeto de desarrollo a no ser que se creen nuevos controles gráficos —nuevos *widgets*—. En el caso de este proyecto han sido necesarios crear varios *widgets*, con lo que finalmente si se ha realizado algo de desarrollo. Entre los *widgets* desarrollados destacan la clase «*Graph*», la clase «*ElementoMenu*» y «*Menu*» —estos controles y el resto desarrollado será objeto de descripción en el capítulo 4 «Realización del proyecto». Los nuevos *widgets* creados se añaden al entorno de desarrollo de igual forma que el resto de código, y en este caso concreto, son incluidos en la carpeta «*Gui*» del proyecto *Keil*:

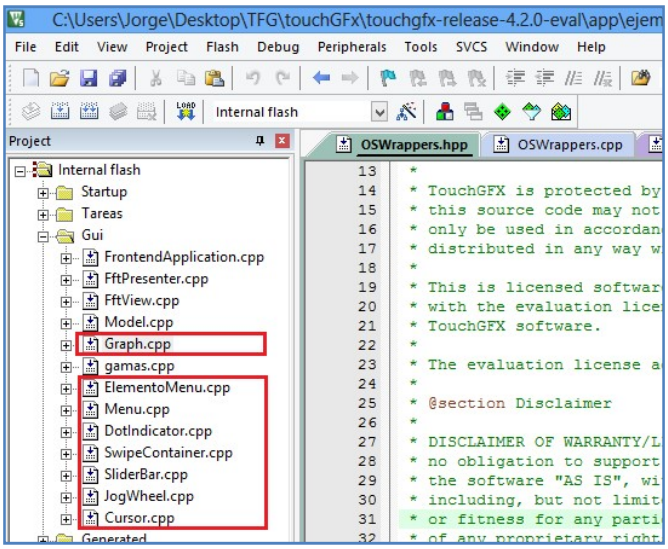


Ilustración 1.6: Inclusión el entorno de desarrollo —Keil— de nuevos widgets creados

Estos nuevos componentes no quedan englobados, físicamente, dentro de la capa *CORE* —dentro de la librería touchgfx_core.lib—, pero debido a la clasificación de los elemento en cada una de las capas, de forma lógica si lo están.

La capa más inferior, la capa *HAL*, es, como su acrónimo indica, una capa de abstracción de hardware. Es decir, esta capa independiza el funcionamiento del entorno *TouchGFX* del hardware concreto utilizado. Además, implementa el concepto de «HAL» de forma verdadera, al contrario que otras librerías como *STM32CubeF4* o *CMSIS* —aunque estas librerías no son comparables con *TouchGFX*, ya que su objetivo no es el desarrollo de una interfaz gráfica, sí adquieren el concepto HAL en su utilización—. En *TouchGFX*, el desarrollador no se tiene que preocupar del hardware que subyace y que está utilizando *TouchGFX* para su funcionamiento, en este sentido es totalmente transparente. La capa *HAL* provee varias funciones a la capa *CORE* para que pueda realizar su actividad. Estas funciones van desde la orientación del display, la funcionalidad del *DMA* —*DMA2D*—, acceso a buffers de display y su manipulación, copiado de regiones de memoria lineal y bidimensional —imágenes— a la función que representa la tarea gráfica *taskEntry*, entre otras. En rara ocasión el desarrollador necesitará acceder a las funciones de la capa HAL directamente, a excepción de la función *taskEntry* que debe ejecutar dentro de una tarea de sistema operativo.

La capa *HAL* está constituida por la instanciación de una sola clase: la clase HAL. Se trata de una clase que implementa el patrón de programación tipo singleton[2] [22] de una manera peculiar. Para la implementación estándar de este patrón de programación, el constructor de la clase ha de ser privado o protegido, con lo que el desarrollador no puede invocarlo directamente y crear el objeto. Este tipo de clases poseen, como atributo interno, una variable objeto, un puntero o referencia de la propia clase —a veces también puede ser una variable estática no perteneciente a la clase—, y un método estático —que puede ser invocado sin tener objeto creado— que sí tiene acceso al constructor. Este método, al ser llamado, primero comprueba que la variable interna, que hace referencia al objeto de la clase, esté vacio, y si es así, llama al constructor creando el objeto en esta variable, para devolverla como valor de retorno. En sucesivas llamadas a este método, al ya existir el objeto creado dentro de la variable interna, se limita a retornar su valor. De esta forma se asegura tener una sola instancia de clase. En el caso de la clase HAL, el constructor es público, con lo que se pueden construir tantos objetos de HAL como se desee, sin embargo para el correcto funcionamiento del entorno *TouchGFX* solo se debe crear un objeto de tipo *HAL*. Para conseguirlo, *TouchGFX* posee un mecanismo de inicialización constituido por la llamada a la función

[2] Patrón de programación donde solo se permite una instancia de clase que lo implementa

touchgfx_init—el proceso de inicialización tanto de hardware como de software, será descrito en el capítulo 3—. Al ser llamada esta función —función perteneciente al espacio de nombres *touchgfx*— crea el objeto de tipo HAL en una variable estática dentro del espacio de nombres —variable llamada *hal* de un tipo derivado de la clase HAL— y su dirección es asignada al puntero estático llamado *instance*, que es una variable miembro de la clase HAL. Esta clase dispone del método estático *getInstance* que devuelve el puntero miembro anterior —la dirección de la instancia de la capa HAL—, necesario para llamar a métodos de la capa HAL —principalmente desde la capa CORE y cuando el desarrollador necesite llamar a alguna función de la capa HAL—.

Si la capa *CORE* no era objeto habitual de desarrollo, la capa *HAL* lo es mucho menos. La capa *HAL* tiene en cuenta los elementos hardware de la placa utilizada —micro y periféricos externos al micro—, y debe existir una clase derivada de la clase *HAL* que implemente la placa concreta a utilizar. Así, por ejemplo, si se utiliza la placa *STM32F429-DISCO*, la clase *HAL* a utilizar es *STM32F4HAL*, con los driver oportunos del *LCD*, del *DMA* gráfico —*DMA2D*— y de la pantalla táctil de la placa disco, que suministra *TouchGFX*, ya que es una de las placas que soporta. En la Ilustración 1.4, aparecen, de forma esquemática, las capas HAL disponibles de los micros soportados para su utilización, bajo el título *SELECT BOARD PACKAGE*, mientras que los driver de dichas capas aparecen bajo el título *SELECT DRIVER PACKAGE*—. Por tanto, la capa HAL, para una placa concreta, está constituida por la clase HAL para el micro de esa placa, más los drivers de los periféricos externos de la placa que *TouchGFX* va a controlar. Si no se utiliza un micro soportado, se ha de desarrollar esta clase derivada, dando cuerpo a varias funciones de la clase HAL —como son *disableInterrupts*, *enableInterrupts*. *configureInterrupts*, *enableLCDControllerInterrupt*, *getTFTFrameBuffer* y *setTFTFrameBuffer*—. No se va a entrar en mayor descripción de la creación de un nuevo tipo de capa HAL, ya que no ha sido necesario realizarlo —debido a que la placa utilizada está soportada por TouchGFX—y la su explicación queda fuera de la pretensión de este documento. Si se dispone de la clase HAL para el micro utilizado, pero no se dispone de los drivers o si se utiliza una placa soportada, pero se modifica algún periférico, como pueda ser el display, se ha de crear el código de dicho driver. En el caso de este proyecto se ha utilizado la placa *STM32F429-DISCO*, que está soportada, pero se instala en ella un nuevo display, con lo que va ha ser necesario suministrar la nueva configuración del display y el driver de la pantalla táctil que viene con él —esto será descrito en el capítulo 3—.

Como sucedía con la capa *CORE*, la capa *HAL* está totalmente compilada en la librería touchgfx_hal.lib para la versión de evaluación, con lo que es necesario añadirla al entorno de desarrollo:

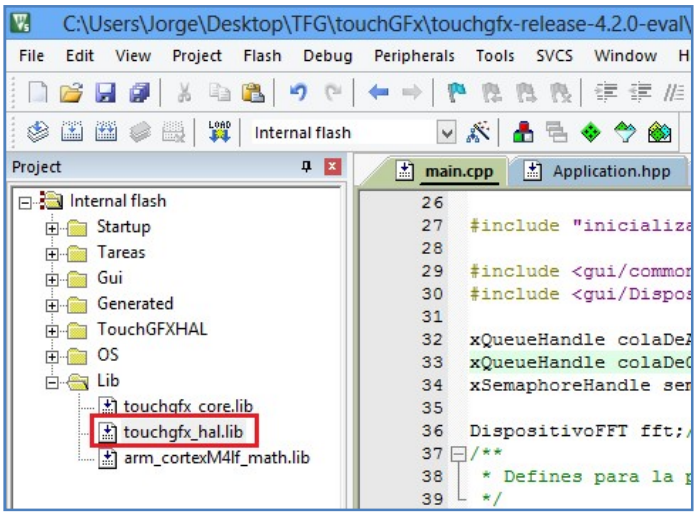


Ilustración 1.7: Inclusión de la capa HAL en el entorno de desarrollo —Keil—

En esta librería está también compilado código ajeno a *TouchGFX* como son la librería CMSIS y la de periféricos estándar del micro *STM32F429I*.

Por último está la capa *OSAL*, que es la capa del sistema operativo. Más que una capa, se trata de una envoltura constituida por la clase *OSWrappers*, cuya misión es la independizar el manejo de dos semáforos, necesarios para la arbitración en el manejo interno del entorno gráfico, del sistema operativo concreto utilizado. Esta clase es añadida al entorno de desarrollo como fichero fuente —ya que se dispone de su código—, y se hace en la carpeta de *OS* del mismo:

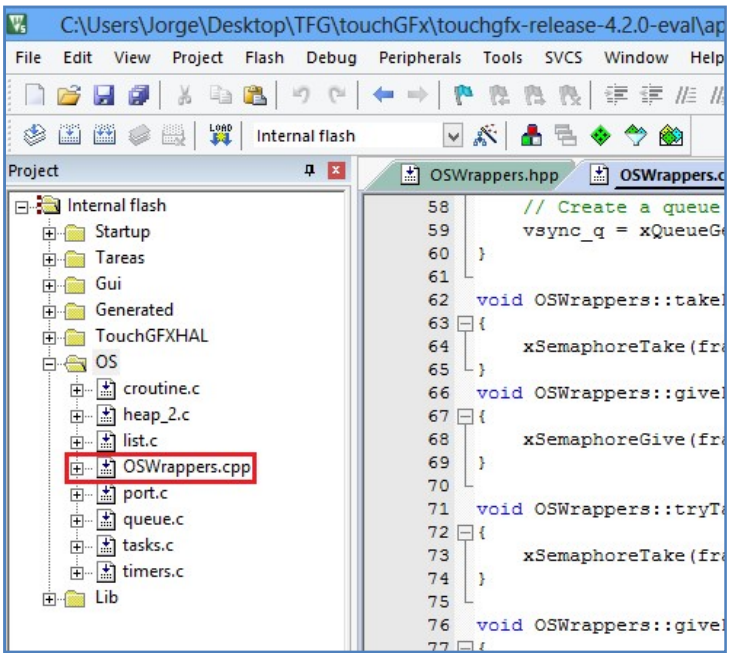


Ilustración 1.8: Inclusión de la capa OSAL en el entorno de desarrollo —Keil—

El sistema operativo que viene instalado con *TouchGFX* es *FreeRTOS*, y a no ser que se desee cambiar el sistema operativo —no es el caso de este proyecto—, esta capa no es objeto de desarrollo. Sin embargo sí se ha de desarrollar código para el sistema operativo ya que la función principal de *TouchGFX* se ejecuta de forma permanente —dentro de un bucle infinito— y la única manera de poder ejecutar otro código —el de la aplicación en sí— es mediante tareas concurrentes del sistema operativo. La descripción del sistema operativo *FreeRTOS*, su configuración para operar con *TouchGFX* y la descripción de la envoltura *OSWarppers*, será objeto del capítulo 3.

1.2.2 Microcontroladores soportados.

Los microcontroladores soportados en la versión 4.2 —la utilizada en este proyecto—de TouchGFX son:

Fabricante	MCU
NXP	LPC1788
NXP	LPC1853
NXP	LPC1857
NXP	LPC4353
NXP	LPC4357
ST	STM32F429
Freescale	K70

Tabla 1: Microcontroladores soportados en TouchGFX 4.2

Los microcontroladores añadidos hasta la versión 4.8 de *TouchGFX* son

Fabricante	MCU
NXP	LPC4088
ST	STM32F439
ST	STM32F469

Tabla 2: Microcontroladores añadidos hasta la versión 4.8 de TouchGFX

1.2.3 Tarjetas directamente soportadas.

Las placas que directamente soporta la versión 4.2 de *TouchGFX*, especificando sus microcontroladores y compiladores utilizados, son:

Placa	MCU	Compilador
FDI uEZGUI-1788-70WVT	NXP LPC1788	IAR
TouchGFX demo board 4.3"	NXP LPC4353, NXP LPC4350	IAR, Keil, GCC
TouchGFX demo board 7.0"	NXP LPC4353	IAR, Keil, GCC
ST STM32429I-EVAL 1	ST STM32f4x9	IAR, Keil, GCC
ST STM324x9I-EVAL-5.7"	ST STM32f4x9	IAR, Keil, GCC
ST STM32F429I-DISCO	ST STM32f4x9	IAR, Keil, GCC
Embedded Artists LPC4357DevKit	NXP LPC4357	IAR, Keil, GCC

Tabla 3: Placas soportadas en la versión 4.2 de TouchGFX

Las placas añadidas, hasta la versión 4.8 de *TouchGFX*, son:

Board	MCU	Compiler
TouchGFX demo board 5.7"	NXP LPC4353	IAR, Keil, GCC
ST STM32F469I-EVAL	ST STM32F469	IAR, Keil, GCC
ST STM32F469I-Discovery	ST STM32F469	IAR, Keil, GCC
Embedded Artists LPC4088DisplayModule	NXP LPC4088	IAR, Keil, GCC

Tabla 4: Placas añadidas hasta la versión 4.8 de TouchGFX

La elección de la placa para desarrollar el proyecto será la *STM32F429I-DISCO*, debido a que es una de las soportadas por el entorno *TouchGFX*´, sus características técnicas y su buena relación calidad-precio. Será descrita en el capítulo 3.

1.3 Distribución de TouchGFX

1.3.1 Distribuciones disponibles

TouchGFX dispone de tres tipos de distribuciones:

Tipo	*Descripción*
Evaluación	Usa librerías precompiladas tanto de la capa CORE como de la capa HAL
Estándar	Usa la librería precompilada de la capa CORE, la capa HAL se suministra con código fuente
Completa	Se suministra el código fuente tanto de la capa CORE como la capa HAL

Tabla 5: Tipos de distribuciones de TouchGFX

La distribución de tipo «Evaluación», no tiene ningún coste, pero aparte de las restricciones mostradas en la tabla anterior, solo permite 150 *Widgets* a la vez en una misma vista —restricción para la versión 4.2, eliminada en la 4.8 y en su lugar, aparece de vez en cuando, el logotipo de *TouchGFX* en movimiento por pantalla—, además se necesita registrarse en la página web del fabricante —*Draupner Graphics* —, y la licencia no permite la creación de dispositivos comerciales. Este es el tipo de distribución utilizada en este proyecto. Al inicio del mismo, la versión era la 4.2 y la actual es la 4.8 —totalmente compatibles—. En ese momento, una vez descargada la versión de evaluación, para descargar una nueva versión, había que adquirir otro tipo de distribución —otro tipo de licencia—. Pero desde que el fabricante de microcontroladores ST lo incluye como posible entrono gráfico para sus dispositivos, está restricción ha desaparecido y esta es la causa por la que solo se haya tenido acceso a estas dos versiones. La versión «Estándar» tiene la restricción de solo tener licencia para fabricar 3000 dispositivos anuales y su coste es de 5000€. La versión «Completa» tiene la restricción de 50000 dispositivos por año y su coste es de 15000€. A parte de estas licencias, para otros escenarios, hay que contactar directamente con *Draupner Graphics* para concretar la licencia y precio concretos.

1.3.2 Compiladores soportados

Para las versiones de la 4.2 a la 4.8 de *TouchGFX*, las utilidades de compilación soportadas son:

1. IAR C/C++ Compiler for ARM.
2. Keil Compiler for ARM.
3. GNU GCC Compiler for ARM.

Para este proyecto se ha utilizado *Keil*. Además, *TouchGFX* suministra un compilador con la distribución, denominado *MinGW*, cuya finalidad es la de compilar los recursos de textos, tipografías e imágenes mediante las utilidades respectivas —ver sección 1.4—.

1.3.3 Estructura de directorios de la librería.

La estructura de directorios del paquete *TouchGFX* es la siguiente:

```
|-- app
|       |-- demo
|       |-- example
|       \-- template
|               \-- EmptyApplication
|-- doc
|       |-- html
|       |-- index.html
|       |-- manual.pdf
|-- touchgfx
        |-- board
        |-- config
        |-- framework
        |-- lib
        \-- os
```

Tabla 6: Estructura de directorios del paquete TouchGFX

Tal y como se muestra en la tabla, La estructura de directorios de la librería se divide en las carpetas «*app*», «*doc*»", y «*touchgfx*». El directorio «*app*» es donde deben ir los nuevos proyectos realizados con *TouchGFX*. De hecho, ya hay proyectos cometidos dentro de esta carpeta. Estos proyectos son las demostraciones —subcarpeta «demo»— que suministra el fabricante a modo de ejemplo de aplicaciones completas —sin la parte de la aplicación final de usuario o *backend* y la comunicación de esta y *TouchGFX*—, y la subcarpeta «*example*», donde aparecen proyectos simples en los que se muestra la utilización de uno o varios *Widgets* por separado, a modo de ejemplo. Bajo la carpeta «*app*», se encuentra también, la carpeta «*template*» —plantilla—, carpeta a partir de la cual se crearán los nuevos desarrollos. Contiene la subcarpeta «*EmtyApplication*», que representa, realmente, el esqueleto de la nueva aplicación. La carpeta «*doc*» contiene la documentación de la librería, tanto en formato *HTML* —index.html y carpeta *html*—, como en *PDF* —manual.pdf—. El directorio «*touchgfx*» es el que realmente contiene los archivos fuente, las librerías y utilidades para crear una nueva aplicación de entrono gráfico *TouchGFX*. Su estructura de subdirectorios es la siguiente:

```
touchgfx
|-- board
|    |-- include
|    \-- source
|-- config
|-- framework
|    |-- include
|    \-- tools
|-- lib
|    |-- board
|    |-- core
|    |-- linux
|    |-- sdl
|    \-- win
\-- os
     \-- FreeRTOS7.6.0
```

Tabla 7: Estructura de directorios de la subcarpeta touchgfx

El directorio *board* contiene los subdirectorios *include*, donde se incluyen archivos de cabecera del fabricante de la placa en cuestión —en este caso la *STM32F429I-DISCO*—, mientras que en la carpeta *source*, se encuentran los archivos fuente BoardConfig.cpp —ver sección 3.5—, para configurar el hardware de la placa para *TouchGFX*, y GPIO.cpp —ver sección 1.15—, para configurar los pines de puerto que actuarán de señales de comprobación de la actividad interna de *TouchGFX*. La carpeta *framework* es donde se localiza, junto con la carpeta *lib*, la esencia de la librería. En la subcarpeta *framework/include*, se localizan lo ficheros de cabecera de todas las clases utilizas en *TouchGFX*. En la subcarpeta *framework/tool*, están las tres utilidades que generan los recursos de textos, tipografías e imágenes para una nueva aplicación —ver sección 1.4—. Los subdirectorios más importantes de la carpeta *lib*, son *board* y *core*. En el subdirectorio *board* se localiza la librería precompilada de la capa *HAL —touchgfx_hal.lib—* que ha de ser añadida al proyecto, mientras que el subdirectorio *core*, contiene la librería precompilada de la capa *CORE— touchgfx_core.lib—*, que ha de ser, igualmente, incluida en el proyecto del compilador —*Keil*—. Finalmente se localiza el directorio *os*, con el subdirectorio *FreeRTOS7.6.0*, que contiene el sistema operativo de tiempo real —ver capítulo 2—.

1.3.4 Estructura de directorios de un nuevo proyecto

La estructura de directorios, en disco, de un nuevo proyecto *TouchGFX* es la siguiente:

```
|-- assets
|       |-- fonts
|       |-- images
|       \-- texts
|
|-- build
|-- config
|       |-- gcc
|       \-- msvs
|
|-- generated
|       |-- fonts
|       |-- images
|       \-- texts
|
|-- gui
|-- platform
|-- simulator
\-- target
```

Tabla 8: Estructura de directorios de un nuevo proyecto TouchGFX

En el directorio *assests*, se colocaran los recursos de fuentes, textos e imágenes que deben ser compilados —convertidos a código c++ en forma de *arrays*— mediante las utilidades de recursos — ver la sección 1.4—. El código generado por estas utilidades de conversión, es localizado bajo la carpeta *generated*, en la subcarpeta correspondiente al recurso convertido, y este código generado ha de ser incluido en el proyecto del compilador —Keil—. En la carpeta *config*, y dentro de *gcc*, se encuentra el archivo de configuración de las utilidades de conversión, llamado *app.mk*. En el directorio *platform*, en su subdirectorio *os*, se encuentra el archivo de configuración del sistema operativo *FreeRTOSConfig.h* —ver sección 2.5—. En el directorio *simulator*, se localiza el archivo main.cpp para el simulador —archivo ejecutable en Windows y Linux que permite simular el aspecto y funcionamiento del entorno gráfico creado— que junto con la directivas *#ifndef SIMULATOR* dentro del resto del código, permiten crear un código paralelo al de la aplicación real para generar el ejecutable de la simulación. Este ejecutable es generado en la carpeta *build*. Debido a la duplicidad de código que supone el crear el simulador, para este proyecto no ha sido utilizado. En la carpeta *target*, se localiza el archivo main.cpp de la aplicación real, así como diverso código de inicialización. En este proyecto, se ha incluido dentro de esta carpeta, los archivos *inicializa_hard.h/c*, que inicializa el generador y señal de pruebas utilizado —ver sección 4.3—, el archivo *system_stm32f4xx.c*, para bajar la frecuencia del sistema a 144 MHz —ver sección 4.2—, y la librería matemática de CMSIS —DSP— *arm_cortexM4lf_math.lib*, para poder utilizar la FFT y otras funciones matemáticas —ver sección 4.5—. Finalmente está el directorio más importante: *gui*. Este, se divide a su vez en los directorios *include* y *src*. En el primero de ellos van todos los archivos de cabecera del entorno gráfico, es decir, los archivos de cabecera del modelo, de la/s vista/s, del/los presentador/es —ver sección 1.9—, y todo el código relacionado con el entrono gráfico, desarrollado por el programador, que considere necesario. Así, para este proyecto, se han

utilizado, dentro de la carpeta gui/include, las subcarpetas *model*, para contener las cabeceras del modelo y el interface que el presentador muestra al modelo; *FFTScreen*, que contienen las cabeceras de la vista y el presentador; *DispositivoFFT*, que contiene la cabecera de la aplicación final de usuario o *backend* —aplicación realmente útil que no pertenece al entorno gráfico, en este caso, la dedicada a adquirir datos y calcular sobre ellos la FFT—; y *common*, que contiene las cabeceras de todos los nuevos controles creados así como las de aquellos que no están integrados en la librería precompilada de *TouchGFX*, pero que han sido conseguidos mediante el código de ejemplo o de la página web del fabricante. Por otro lado, en la carpeta gui/src, se localizan carpetas con el mismo nombre que las del directorio *gui/include*, pero su contenido son los archivos fuente de esas cabeceras.

1.4 Crear un nuevo proyecto TouchGFX

Para crear una nueva aplicación en *TouchGFX* se ha de partir de la plantilla que se facilita para ello, dentro de *app/template/EmptyApplication*, como ya ha sido anticipado en la sección 1.3. Esta plantilla contiene toda la configuración necesaria para todas las placas soportadas, para todas las aplicaciones de compilación soportadas —*IAR*, *Keil* y *GCC*— y para el simulador —*MinGW* o *MSVS*—. Los pasos generales a seguir para la nueva aplicación, son los siguientes:

1) Copiar app/*template/EmptyApplication* a otra carpeta, dentro del directorio *app*, por ejemplo, llamada *Proyecto*, y dentro de ella, renombrar la carpeta *EmptyApplication* con un nombre descriptivo del proyecto, como por ejemplo, *FFTAplicacion*, quedando la nueva aplicación en la ruta *app*/*Proyecto*/*FFTAplicacion*.
2) Borrar todas las carpetas bajo la ruta *app*/*Proyecto*/*FFTAplicacion* /*target* que sean distintas al fabricante de la placa utilizada, en el caso concreto de este proyecto, borrar todas las carpetas que no sean *ST*.
3) Dentro de *target/ST*, borrar todas las carpetas que sean distintas a la que hace referencia a la placa concreta utilizada, en el caso de este proyecto, borrar todas las carpetas que no sean STM32F429I-DISCO.
4) Dentro de *target/ST/ STM32F429I-DISCO*, borrar todas las carpetas distintas a la que haga referencia a la utilidad de compilación contemplada, en este caso, borrar todas las carpetas distintas de *Keil*.
5) Renombrar las carpetas *app*/*Proyecto*/*FFTAplicacion*/*gui*/*src*/*template_screen* y *app*/*Proyecto*/*FFTAplicacion*/*gui*/*include*/*gui*/*template_screen* al nombre de pantalla inicial —par vista-presentador— a utilizar, en el caso de este proyecto, ambas al nombre de *FFT_screen*.
6) Renombrar los archivos fuentes *TemplatePresenter.cpp*, *TempleteView.cpp* y las cabeceras *TemplatePresenter.hpp*, *TempleteView.hpp*, de los directorios anteriores, a los nombres de presentador y vista principal, en el caso de este proyecto a *FftPresenter.cpp*, *FftView.cpp*, *FftPresenter.hpp* y *FftView.hpp*, respectivamente.
7) Modificar el archivo *FrontendHeap.hpp*, para especificar estos archivos en la asignación estática de memoria —ver sección 1.10—.
8) Añadir a la carpeta *assets/fonts* todos los archivos de fuentes utilizado —en este proyecto *RobotoCondensed-Regular.tft*—; añadir a la carpeta assets/images, todas las imágenes utilizadas —en formato *bmp* o *png*—; y añadir en la carpeta *assets/texts,* todas las cadenas de texto utilizadas, especificadas en el archivo *Excel texts.xlsx* —seguir leyendo esta sección para más detalle—.
9) Compilar los recursos con el comando: *make -f simulator/gcc/Makefile clean assets*, mediante el compilador *MinGW*, suministrado por *TouchGFX*, previo posicionamiento en el directorio del proyecto,

en este caso en *app*/*Proyecto*/*FFTAplicacion*. Esta compilación creará los archivos fuentes de los recursos en la carpeta *app*/*Proyecto*/*FFTAplicacion*/*generated*.

10) Añadir los recursos compilados como ficheros fuente, al proyecto de la aplicación de compilación, en este caso, al proyecto *Keil*.
11) Incluir la sentencia *#include <BitmapDatabase.hpp>* —base de datos de imágenes— en todos los archivos fuentes de las vistas.
12) Incluir la sentencia *#include <texts/TextKeysAndLanguages.hpp>* —identificadores de todas las cadenas— en todos los archivos fuentes de las vistas.
13) En el archivo de cabecera de cada vista, incluir sentencia del estilo *#include <touchgfx/widgets/….>* que hagan referencia a los *widgets* utilizados en esa vista.
14) Incluir en el archivo fuente del presentador, la cabecera de la vista correspondiente, en el caso de este proyecto, *#include <gui/ FFT_screen / FftView.hpp>*
15) Si se necesita algún tipo derivado —como int16_t— en el presentador o la vista, incluir en el archivo de cabecera del presentador o la vista, la sentencia *#include <touchgfx/hal/Types.hpp>*.
16) Incluir en el archivo fuente del presentador la cabecera *#include <gui/model/Model.hpp>*.
17) Crear el archivo *Model.cpp*, emplazarlo en *gui/src/model* y añadirle la cabecera: *#include <gui/model/Model.hpp>*.
18) Si se necesita algún tipo derivado —como int16_t— en el modelo, incluir en el archivo de cabecera del modelo *#include <touchgfx/hal/Types.hpp>*.
19) Poner en el archivo fuente del modelo la cabecera *#include <gui/model/ModelListener.hpp>*.
20) Desarrollar toda la aplicación en sí, añadiendo los pares vista-presentador necesarios, así como nuevos archivos fuente y cabeceras que contengan el código necesario para obtener la utilidad buscada.

Todos estos pasos son básicos e importantes, sin embargo, hay que prestar especial cuidado a los pasos del 16 al 19, ya que en el directorio plantilla de aplicación que suministra *TouchGFX*, no existe fichero fuente del modelo, de hecho, en casi ningún ejemplo suministrado existe este archivo.

En este directorio plantilla, dentro de la carpeta target, está el archivo de proyecto *Keil —uvproj—*. La estructura de este proyecto, ya dentro de la aplicación *Keil*, tiene la siguiente apariencia:

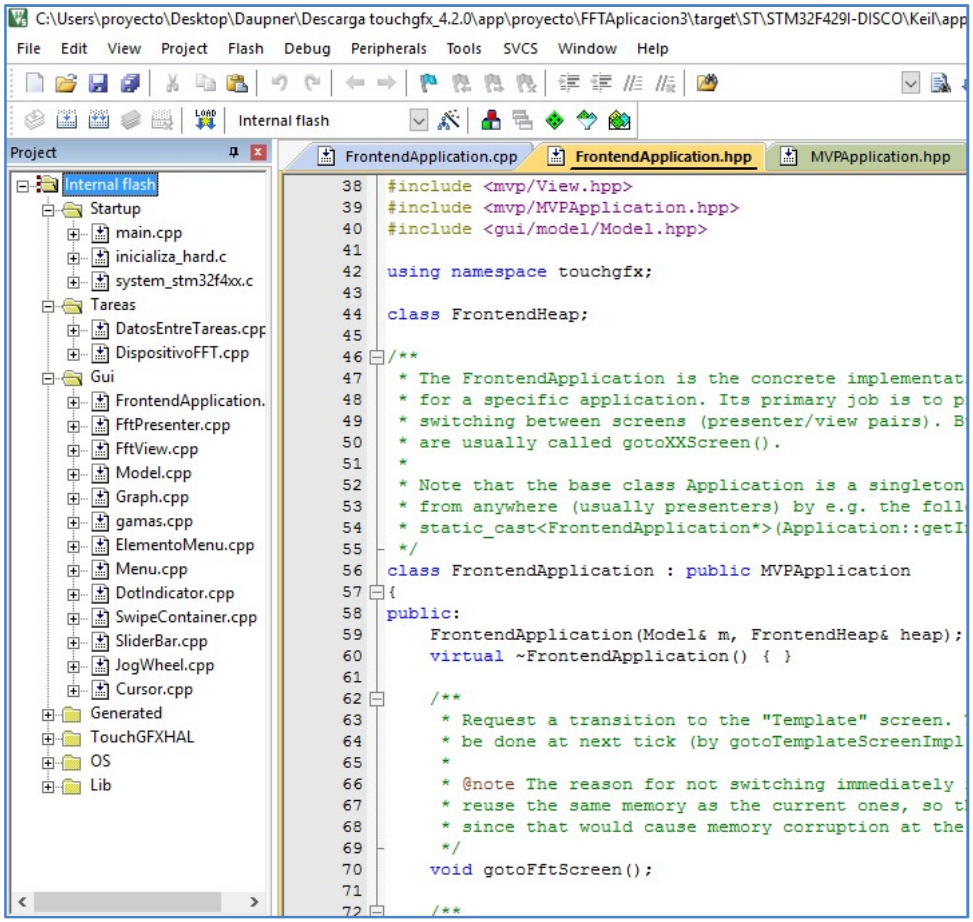


Ilustración 1.9: Estructura de carpetas lógicas dentro del proyecto Keil

Consta de las carpetas lógicas *Startup*, donde se encuentra el archivo *main* y otros que el desarrollador considere de inicialización; *Gui*, donde están los ficheros fuente de las vistas, de los presentadores, del modelo, de la clase que controla en patrón *MVC —FrontendApplication—* y de todos aquellos archivos que el desarrollador considere oportunos —generalmente de nuevos controles gráficos creados—; *Generated*, que contiene los archivos fuente de los recursos compilados; *TouchGFXHAL*, que contiene los archivos fuentes *GPIO.cpp* —ver sección 1.15— y *BoardConfiguration.cpp* —ver sección 3.5—; *OS*, que contiene los archivos fuente del sistema operativo *FreeRTOS*; y *Lib*, que contiene las librerías compiladas añadidas al proyecto, en este caso, contiene los archivos touchgfx_core.lib —capa CORE de *TouchGFX*—, touchgfx_hal.lib —capa HAL de *TouchGFX* — y arm_cortexM4lf.lib —DSP-CMSIS little endian—. También se puede añadir carpetas lógicas nuevas, en el caso concreto de este proyecto se ha añadido la carpeta *Tareas*, que contiene los archivos fuente *DispositivoFFT.cpp*, que encapsula la clase que constituye la aplicación final de usuario o *backend*, que se ejecuta de forma concurrente con *TouchGFX* gracias a *FreeRTOS*, y *DatosEntreTareas.cpp*, que contiene los arrays que comparten las dos tareas: la del entorno gráfico y la de la aplicación final de usuario.

1.4.1 Recurso de imágenes gráficas

Existe, por cada archivo de imagen puesto en el directorio *assets/images*, un archivo fuente con el mismo nombre pero con extensión *cpp* y alojado en *generated/images* —por ejemplo, para el archivo de imagen ani_01.png se generará un archivo fuente ani_01.cpp—. Dentro de cada uno de estos archivos está definido un array con los datos numéricos que representan los puntos de la imagen. Este array se nombra como «extern const unsigned char _<nombre archivo imagen sin extensión>».

Para manejar todos estos archivos de imágenes compiladas, está, como archivo de base de datos de los bitmaps, el archivo «*BitmapDataBase.cpp*». En este archivo se define el array de nombre «*bitmap_database*», array de estructuras de tipo «BitmapData» que sirven para contener características de cada imagen, por lo que este array albergará tantas entradas como bitmaps haya. Para el manejo de los elementos de este, se declaran en el archivo de cabecera —*BitmpaDataBase.hpp*— unos identificadores constantes con el formato de nombre «BITMAP_<nombre archivo bitmap>_ID», cuyo valor es el necesario para servir de índice de este array y obtener los datos referentes al bitmap referente al identificador.

En el archivo «*BitmapDataBase.cpp*» se definen también las funciones «*BitmapData* getInstance()*», que devuelve el array «*bitmap_database*» y «*getInstanceSize()*» que devuelve el tamaño del array —número de bitmaps—.

Desde el punto de vista del programador, lo que tiene que conocer es que se han de añadir al proyecto todos los archivos fuente que corresponden a las imágenes compiladas, así como el archivo fuente que representa la base de datos de imágenes —*BitmapDatabase.cpp*— y que es generado al compilar los recursos —paso 9 en el inicio de la sección 1.4—.

1.4.2 Recurso de fuentes de texto

Otro recurso a utilizar son las fuentes gráficas de texto. Para ello, se alojaran en el directorio *assets/fonts*, todos aquellas fuentes a utilizar. Por ejemplo, en este proyecto solo se ha utilizado una fuente: RobotoCondensed-Regular, así que, se ha incluido el archivo RobotoCondensed-Regular.ttf en ese directorio. Al compilar los recursos —paso 9 en el inicio de la sección 1.4—, se crean, en la ruta *generated/fonts/src*, una serie archivos generales, para todas las fuentes tratadas, y unos archivos específicos para cada fuente y tamaño de la misma —todo esto se determina en la siguiente sección 1.4.3, al concretar las cadenas de texto utilizadas—. Los archivos generales creados son: *ApplicationFontProvider.cpp* y *FontGetters.cpp*. Ambos contienen la definición de los arrays que representan todos los tipos de fuente con su tamaño asociado utilizados, y como se accede a ellos. El desarrollador no tiene que preocuparse del contenido de estos archivos, ya que solo los usará *TouchGFX* de forma interna, lo único que ha de hacer es incluirlos en el proyecto de la aplicación de compilación utilizada —en este caso, añadiendo estos archivos a la carpeta lógica «Generated» del proyecto Keil—. Los archivos específicos, por cada fuente y su tamaño asociado, tienen el siguiente formato de nombre: Font _<nombre fuente>_<tamaño>_<bits renderizado>.cpp, Table_<nombre fuente>_<tamaño>_<bits renderizado>.cpp y kerning_<nombre fuente>_<tamaño>_<bits renderizado>.cpp. En el primero de ellos está el array que contienen la información gráfica de cada letra de la fuente y tamaño utilizado en las cadenas de texto —solo contiene las fuentes de letras utilizadas, no todas las disponibles para esa fuente, lo que ahorra espacio en ROM—. El segundo archivo, contiene un array con información de cada grafía guardada en el archivo anterior, como qué letras están guardadas y como se accede ellas. El tercer archivo contiene información de espaciado entre grafías, ya que no es un valor constante. Por ejemplo, si se hubiese utilizado el archivo de fuentes antes mencionado, para un tamaño

de letra 10 y con renderizado de 4 bits, el nombre de los archivos respectivo sería: *Font_RobotoCondensed_Regular_10_4bpp.cpp*, *Table_RobotoCondensed_Regular_10_4bpp.cpp* y *Kerning_RobotoCondensed_Regular_10_4bpp.cpp*. Desde el punto de vista del desarrollador, lo único que ha de tener en cuenta, es que se generan estos tres archivos específicos por cada fuente y tamaño de letra y que, al igual que con los archivos generales, and de ser incluidos en el proyecto de la aplicación de compilación. Al alojar el archivo de fuentes gráficas en el directorio *assets/fonts*, solo se ha especificado e tipo de la fuente, no el tamaño. Esto es algo que se hace al definir las cadenas de texto utilizadas en *TouchGFX* —siguiente sección 1.4.3—. Cada cadena de texto, además de la secuencia de letras que representa, tiene asociada un tipo de fuente y un tamaño.

1.4.3 Recurso de cadenas de texto

En *TouchGFX*, cada cadena de texto, aparte de la información que representa la secuencia de letras, tiene asociado una fuente tipográfica y un tamaño concretos. Así, cuando se utilice una cadena determinada, implícitamente se está especificando su fuente y tamaño. Para crear las cadenas, *TouchGFX* facilita un archivo *Excel*, llamado *texts.xlx*. Esta hoja de cálculo tiene dos pestañas, una define las tipografías usadas y se denomina «Typography» y la otra define las cadenas de texto utilizadas en el código y se llama «Translation». La pestaña «Typography» especifica las tipografías usadas en la aplicación y que únicamente se utilizan como un «alias» que representa el tipo de letra y su tamaño para ser referenciado en la otra pestaña «Translation». Especifica los siguientes valores:

- *Typography Name*: alias que se le dará al tipo de letra con un tamaño concreto para ser referenciado en la otra pestaña «Translation».
- *Font*: nombre de la fuente usada en la tipografía. Esta/s fuente/s serán las que aparezcan en el directorio «*asses/font*». Deben incluir el nombre completo de la fuente.
- *Size*: tamaño de la tipografia
- *Bpp*: numero de bits por pixel usados para representar a la fuente. Valores válidos son: 1, 2, 4 y 8.

La apariencia de esta pestaña, y que es la utilizada en este proyecto, es la siguiente:

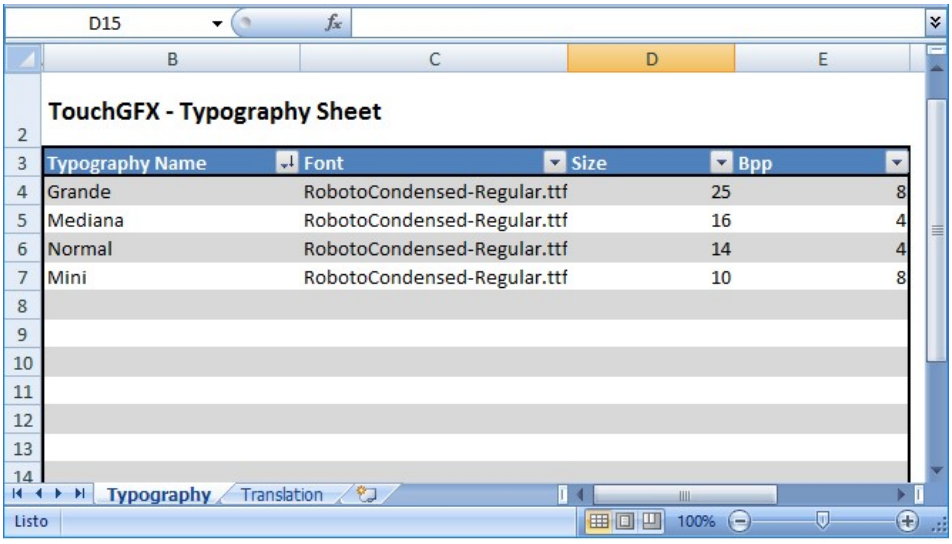
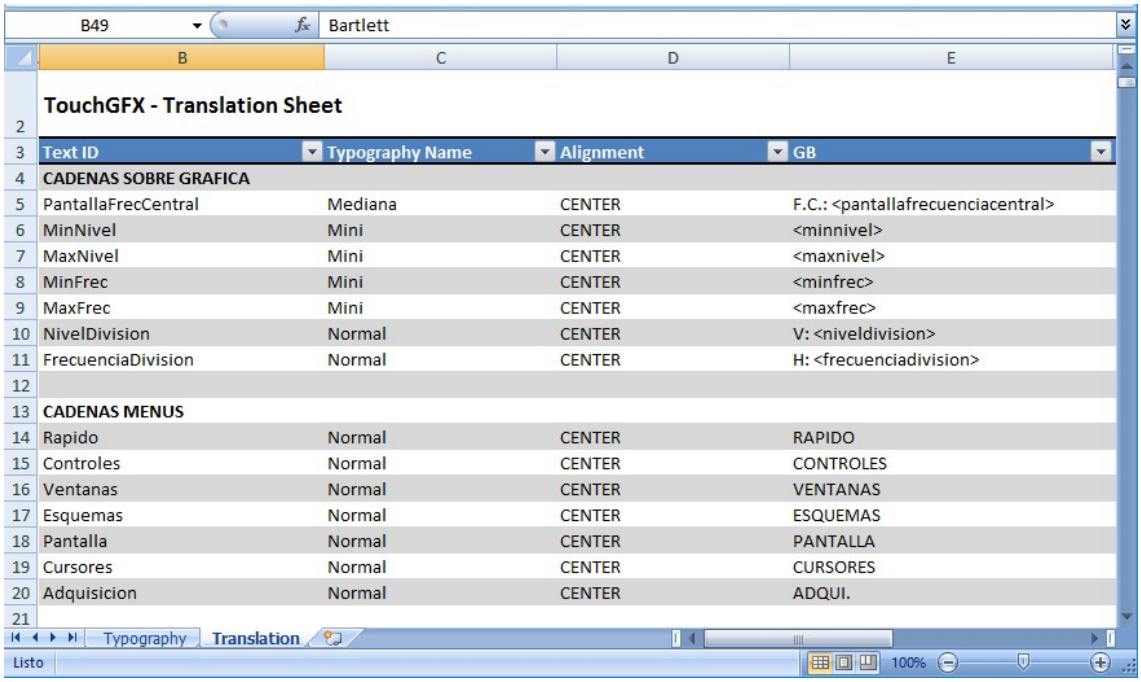


Ilustración 1.10: Pestaña Typography del archivo texts.xlsx usado en el proyecto

La pestaña «Translation» permite definir las cadenas que se van a utilizar en el código. Cada fila de esta hoja representa una cadena, que incluye tipografía, alineamiento y soporte para traducción en diferentes idiomas. Contiene las siguientes columnas:

- *Text ID*: es el identificador de la cadena en el código del desarrollador. Para referenciar la cadena en el código, hay que incluir la cabecera «*TextKeysAndLanguages.hpp*» en el archivo fuente de la vista. Este archivo de cabecera, define todas las cadenas y lenguajes. Contiene dos estructuras; una referente a las cadenas «*TEXTS*» y otra a los idiomas soportados «*LANGUAGES*». Todos los identificadores aparecerán como elementos de la estructura «*TEXTS*», con el identificador en mayúsculas y precedidas del prefijo «*T_*». Así mismo, los lenguajes consistirán en 1 a 3 caracteres en mayúsculas y aparecerán como elementos de la estructura «*LANGUAJES*». Para utilizar las cadenas, se llamarán los métodos de adquisición de texto de los objetos, con argumento «*TypedText(<identificador de la cadena>)*», por ejemplo: *btnToggle.setLabelText(TypedText(T_START))*; donde «*btnToggle*» es un objeto de tipo botón al que se le quiere asignar como texto la cadena identificada por *T_START* —que es uno de los elementos de la estructura «*TEXTS*» en el archivo de cabecera «TextKeysAndLanguages.hpp», como se ha mencionado antes—.
- *Typography Name*: especifica que tipografía usará por defecto la cadena —alias que en la hoja «*Typography*» se dio a la fuente y su tamaño—.
- *Alignment*: especifica el alineamiento de la cadena que puede tomar los valores de *LEFT*, *RIGHT*, y *CENTER*.
- Columna lenguaje: será una columna cuya cabecera será de 1 a 3 caracteres especificando el lenguaje en el que se encuentra la cadena. El contenido de esta columna es la cadena en sí.
- Tipografía específica del lenguaje: la cabecera de esta columna será «idioma-TYPOGRAPHY», ejemplo: *GB-TYPOGRAPHY* —siendo GB referente a idioma inglés—. Con esta columna sobrescribimos la tipografía usada por defecto para un lenguaje específico. Los valores de esta columna deben ser valores existentes de tipografías en la hoja «*Typography*». Esto es útil si el idioma usa caracteres que no está incluidos en la fuente de la tipografía por defecto, por ejemplo, los caracteres chinos. Si el contenido de esta columna se deja vacio, la tipografía usada para la cadena es la por defecto —columna «*Typography Name*»—.
- Alineamiento especifico del lenguaje: se puede sobrescribir el alineamiento por defecto, añadiendo una columna llamada «*language-ALIGNMENT*», por ejemplo, *GB-ALIGNMENT*. El contenido es el mismo de la columna *Alignment*, y si se deja vacio, se usa el alineamiento por defecto.

El aspecto de esta pestaña, y que es la utilizada en este proyecto, es el siguiente:



B49 | Bartlett

	B	C	D	E
2	**TouchGFX - Translation Sheet**			
3	**Text ID**	**Typography Name**	**Alignment**	**GB**
4	**CADENAS SOBRE GRAFICA**			
5	PantallaFrecCentral	Mediana	CENTER	F.C.: <pantallafrecuenciacentral>
6	MinNivel	Mini	CENTER	<minnivel>
7	MaxNivel	Mini	CENTER	<maxnivel>
8	MinFrec	Mini	CENTER	<minfrec>
9	MaxFrec	Mini	CENTER	<maxfrec>
10	NivelDivision	Normal	CENTER	V: <niveldivision>
11	FrecuenciaDivision	Normal	CENTER	H: <frecuenciadivision>
12				
13	**CADENAS MENUS**			
14	Rapido	Normal	CENTER	RAPIDO
15	Controles	Normal	CENTER	CONTROLES
16	Ventanas	Normal	CENTER	VENTANAS
17	Esquemas	Normal	CENTER	ESQUEMAS
18	Pantalla	Normal	CENTER	PANTALLA
19	Cursores	Normal	CENTER	CURSORES
20	Adquisicion	Normal	CENTER	ADQUI.
21				

Typography | Translation

Listo 100%

Ilustración 1.11: Pestaña Translation del archivo texts.xlsx usado en el proyecto

Sin embargo, la creación de estas cadenas solo permite especificar textos constantes. Para poder especificar textos variables, *TouchGFX* provee lo que se denominan cadenas comodines dentro de la definición de la cadena constante. Estas se especifica con <*>, donde * representa un texto de ayuda opcional que no será incluido en la cadena de texto resultante. En la ilustración anterior se muestran varias cadenas variables. Es posible tener hasta dos comodines en una cadena de texto. Si se usan cadenas para más idiomas, estas deben tener el mismo número de comodines que la cadena del lenguaje principal. Este/os valor/es comodín se insertan en tiempo de aplicación en el código C++. Un ejemplo del uso sería: «La temperatura es <insertar_temperatura>°». Los caracteres, que se guardan en memoria ROM, son solo aquellos que se usan en una cadena de un tipografía concreta. Por ello, si se usan comodines, hay que asegurarse que los caracteres de los posibles valores de estos comodines existen en alguna cadena. Para ello es aconsejable hacer una cadena «tonta» que contenga los caracteres que pueda tomar el comodín, por ejemplo el texto "tonto": "-0123456789". Esto asegura que el comodín pueda representar cualquier carácter numérico. En el texto aún se pueden utilizar los caracteres '<' y '>' sin que sean interpretados como límites de un comodín, para ello se han de preceder del una contrabarra, por ejemplo: "\<esto no es una cadena comodín\>".

1.5 Controlador de display en TouchGFX y temporizaciones

El conocimiento de los tiempos involucrados en el ciclo de actualización del *display* es importante, ya que el funcionamiento de *TouchGFX* es totalmente dependiente de él. Es cierto que todo este proceso es transparente al desarrollador, pero hay situaciones en las que se deben solventar problemas de refresco de pantalla y sí se debe conocer este funcionamiento. En *TouchGFX* hay dos procesos básicos: la actualización del *display* y la actualización del buffer secundario. La actualización del *display* es realizada constantemente por parte del controlador hardware del *TFT* —en el micro *STM32F429* es el *LCD-TFT*—, incluso si la imagen no cambia. Debido a que el *display* no tiene memoria —en el caso concreto de este proyecto—, el controlador debe pasar, de forma periódica, la información gráfica contenida en la RAM externa al *display*. Esta área de memoria constituye el buffer primario gráfico. Por otro lado, existe un proceso software gráfico que testea los eventos producidos y actualiza, en consecuencia, un área de memoria que en principio es igual al buffer primario. Esta otra área es el buffer secundario gráfico y es intercambiado con el primario cuando este proceso realiza modificaciones sobre el buffer secundario, que provoquen que sea diferente al primario. Entonces el buffer secundario pasa a ser el primario y viceversa. Todas estas operaciones tienen que estar sincronizadas, y esta sincronización se realiza acorde con las temporizaciones del controlador TFT:

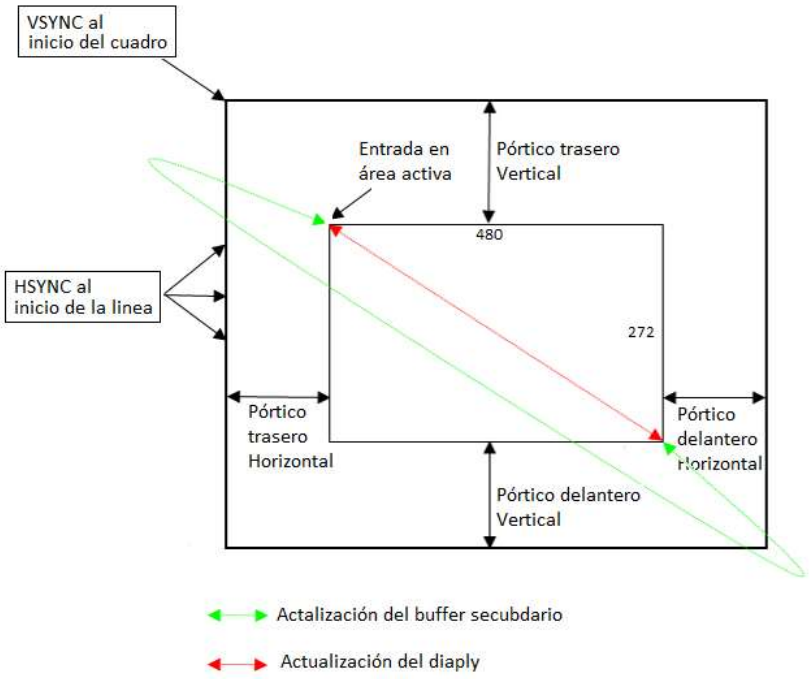


Ilustración 1.12: Ciclo de actualización del Display

En esta ilustración [1] se muestra el intervalo de tiempo en el que se produce el refresco del *display* con el buffer primario mediante el controlador *TFT*, mostrado como una flecha doble de color rojo. También se muestra el intervalo de tiempo en el que se modifica el buffer secundario por parte del proceso software gráfico, mostrado como doble flecha verde. Cuando se inicia el proceso de actualización del *display*, el controlador lo marca mediante el sincronismo vertical. Sin embargo, este no es el momento en que se inicia el refresco del *display*. Verticalmente, antes del inicio de cada cuadro, existe un lapso de tiempo en el que no se está dibujando la imagen. Este periodo es el pórtico trasero vertical. Al finalizar el cuadro, verticalmente, existe otro intervalo en el que no se dibuja; es el pórtico delantero vertical. Al inicio de cada línea y al finalizarla, están los pórticos trasero y delantero horizontal, respectivamente. Es decir, que el tiempo en el que se está dibujando la imagen —

zona activa— es solo un intervalo del proceso de actualización del display. Sin embargo, aunque no se pueda acceder al buffer primario en el momento que el controlador *TFT* está en la zona activa, el proceso software puede estar procesando información. De hecho, estos son los pasos que este proceso lleva a cabo en cada actualización:

1. Llamar al método *beginFrame* del objeto HAL, para, probablemente[3], realizar tareas iniciales, como intercambiar los buffers si estos son distintos —se han producido modificaciones en el buffer secundario en el ciclo de refresco anterior—.
2. Gestionar las temporizaciones. Esto consiste en llamar al método *Application::handleTickEvent* que a su vez llama a los métodos homónimos de los controles que se han registrado para recibir eventos de temporización, así como el método homónimo de la vista activa —ver sección 1.11.3— y previamente a todo ello, el método *tick* del modelo —ver sección 4.6.3—. Este paso puede que implique actualización en el buffer secundario.
3. Testear los toques en pantalla y enviar estos eventos a la aplicación. Este paso puede requerir la actualización del buffer secundario.
4. Testear botones hardware —si se gestionan a través de *TouchGFX*— y enviar los eventos correspondientes a la aplicación. Este paso puede requerir actualización del buffer secundario.
5. Comenzar a realizar las actualizaciones gráficas requeridas en los pasos 2, 3 y 4 en el buffer secundario —escritura del buffer secundario—.
6. Esperar a que las actualizaciones terminen.
7. Llamar al método *endFrame* para, probablemente[3], ejecutar el método *waitForVSync*, de la clase *OSWrappers* —ver sección 2.6— que deja la tarea software gráfica en estado de espera hasta el siguiente sincronismo vertical.

Para entender la sincronización entre el controlador TFT y la tarea software gráfica, el fabricante presenta el siguiente cronograma:

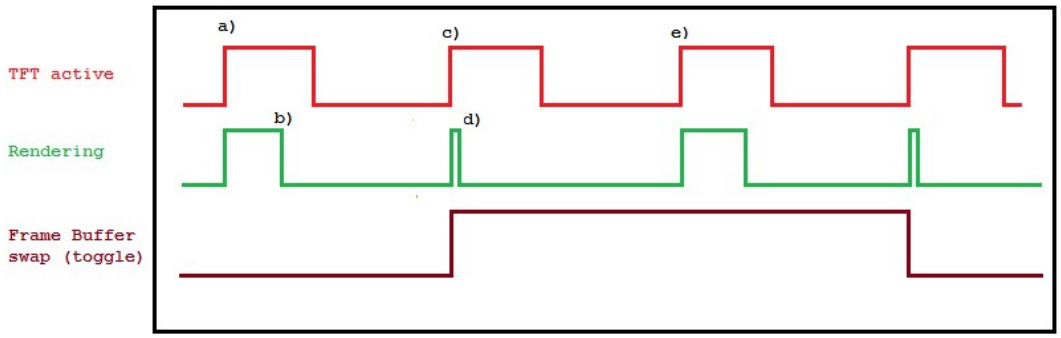


Ilustración 1.13: Cronograma de temporizaciones del controlador TFT y la tarea software

La gráfica de color rojo representa la temporización del controlador TFT o refresco del display. En ella se han marcado los puntos a), c) y e), que son los instantes en los que el controlador TFT entra en la zona activa de la imagen. El estado de temporización del controlador permanece a nivel alto mientras esté en esta zona. La temporización de la tarea software es mostrada en la gráfica de color verde —etiquetada como «Rendering»—. Cuando el TFT entra en la zona activa en a), la tarea software comienza su labor, que en este caso, es menor en tiempo que la del TFT, y en el punto b) entra en estado de espera hasta el siguiente sincronismo vertical. En este ejemplo se ha de suponer que la tarea software ha modificado el buffer secundario. Cuando el TFT entra en el punto c), se activa la tarea software, y lo primero que hace es intercambiar los buffers —gráfica de color

[3] Se utiliza el adjetivo «probablemente» ya que no se tiene acceso al código —está compilado en la versión de evaluación—.

marrón— ya que en el ciclo anterior de refresco había modificado el buffer secundario. En este caso, se supone que la tarea software no realiza ninguna modificación en el buffer secundario e inmediatamente entra en estado de espera en el punto d). En el siguiente ciclo de refresco, punto e), como en el ciclo anterior no se ha producido ninguna modificación en el buffer secundario, no se produce intercambio de buffers. A partir de aquí, para el caso de este ejemplo, el proceso se repite. Este ejemplo muestra la situación en que la temporización del proceso software es menor a la temporización necesaria para refrescar el display. En este caso, la velocidad de actualización de cuadros de la imagen coincide con la velocidad de refresco del display. Sin embargo, puede ocurrir que la tarea software tarde más en su labor que el tiempo que emplea el TFT en refrescar el display. En este caso la velocidad de actualización de cuadros será varas veces menor a la de refresco del dsplay. Si esto ocurre de vez en cuando, no hay problema, pero si ocurre de forma continuada para imágenes en constante movimiento, como en el caso de una gráfica en la que constantemente están variando sus valores —caso de este proyecto—, no es tolerable. Esta situación queda plasmada en el siguiente cronograma ejemplo, facilitado por el fabricante:

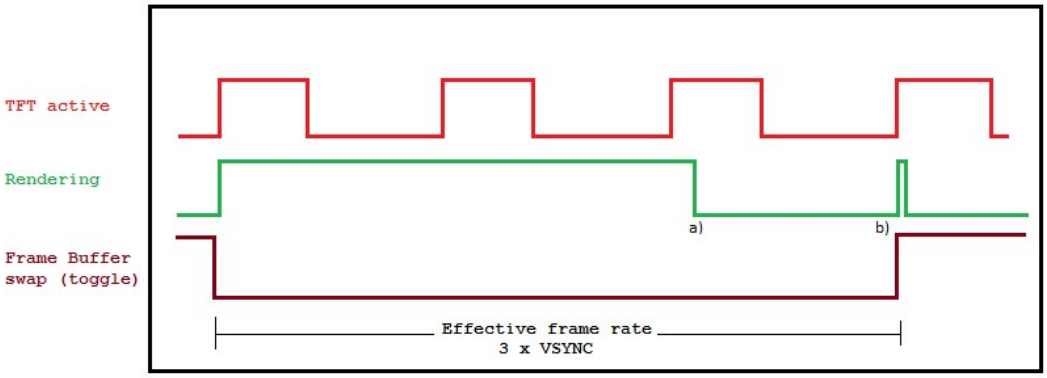


Ilustración 1.14: Actualización de cuadros de imagen menor al refresco del display

En este caso la tarea software, que comienza al inicio del refresco del display, necesita de tres periodos de refresco para completar su labor en el punto marcado como a). Como este punto se encuentra ya dentro del refresco de display, no se puede realizar el intercambio de buffers y se debe esperar al siguiente ciclo, punto b), para ejecutarlo. Esto hace que la velocidad de actualización de la información gráfica sea tres veces menor a la velocidad de refresco del display. Si la cantidad de información a actualizar produce de forma continuada este efecto, la solución es bajar la velocidad de refresco del display para que la actualización de la información se produzca en menos ciclos de refresco. Esta situación se muestra en el siguiente cronograma:

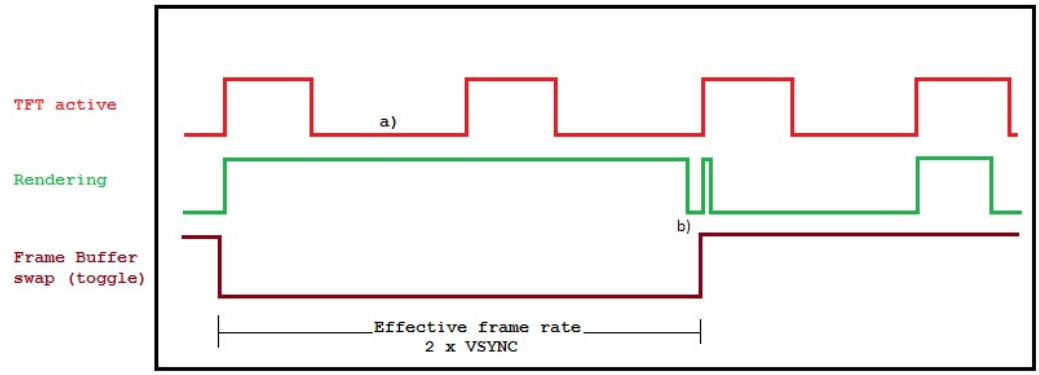


Ilustración 1.15: Reducción de la velocidad de refresco para aumentar la de actualización

Aunque cada ciclo de refresco es ahora mayor, dos de estos ciclos, requeridos para la actualización de la información, suponen menos tiempo que los tres ciclos de refresco anteriores, a pesar que cada uno de ellos dure menos. Para conseguir menor velocidad de refresco, se han de aumentar los pórticos.

Hasta ahora se ha mostrado la temporización de actualización de la información bajo la tarea software gráfica, pero esta delega parte de su labor al *DMA* —concretamente al *DMA2D* para microcontroladores *ST*, el caso de este proyecto—. Esta delegación se produce cuando las modificaciones del buffer secundario requieren ser realizadas con información gráfica guardadas en memoria —generalmente en ROM—. Cuando no sea así, ha de ser la tarea software la que realice las modificaciones directamente. En el siguiente cronograma facilitado por el fabricante, se muestra el entrelazado entre la tarea software y el *DMA*:

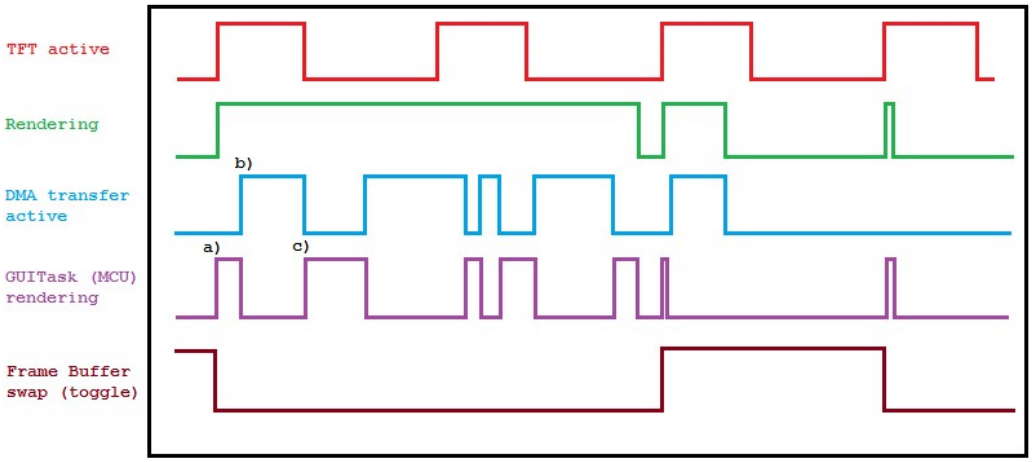


Ilustración 1.16: Entrelazado de tarea software y DMA para la actualización gráfica

Como en ilustraciones anteriores, la gráfica de actualización de la información se muestra en verde. Sin embargo es el resultado —la suma— de la tarea software —mostrada en morado, bajo la etiqueta de GUITask—, y el DMA —mostrado en color azul—. Al iniciar el ciclo de refresco del display —gráfica roja—, el control de la actualización lo tiene la tarea software, punto marcado como a). Cuando necesita actualizar el buffer secundario con información contenida en memoria, delega en el DMA, punto marcado como b). Cuando la trasferencia DMA termina, y si la actualización del buffer secundario no se realiza como trasferencia de memoria, la tarea software vuelve a tomar el control —punto marcado como c)—. Este intercalado entre tarea software y DMA sucede durante todo el proceso de actualización de la información gráfica. La causa del uso de DMA, es la de liberar al núcleo del micro de carga y de esta forma aumentar la rapidez en el procesamiento. Sin embargo, cuando el hardware no soporte transferencias DMA bidimensionales —no es el caso de este proyecto—, será la tarea software la que realice todo el trabajo.

Como se ve en esta última gráfica, y ya que tanto la tarea software como el DMA acceden a un mismo recurso —el buffer secundario—, se necesita un sistema de arbitración de tipo semáforo, que cuando uno de ellos coja su testigo, el otro quede a la espera hasta ser devuelto. Concretamente, cuando la tarea software requiera los servicios del DMA, activará a este y eliminará el testigo del semáforo, bloqueándose a la espera de testigo. El DMA, mediante la interrupción que marca el fin de la transferencia, generará el testigo para desbloquear a la tarea software. De forma similar, ya que la tarea software es la encargada de intercambiar los buffers que usa el controlador TFT, y este intercambio solo puede ser realizado en momentos determinados, se necesita un sistema para que el controlador TFT informe a la tarea software del momento exacto. Esto puede ser realizado con otro semáforo, donde el controlador TFT, mediante la interrupción que se genera al entrar en la zona activa, facilite el testigo, la tarea software lo recoja, realice el intercambio de buffers si es posible y necesario, realice el resto de su labor —junto con el DMA— y se bloquee a la espera de otro testigo generado por el controlador TFT. Estos dos semáforos, así como la forma de acceder a ellos, están definidos en el par de archivos OSWrappers.hpp/cpp, que constituyen la capa OSAL —ver sección 2.6—.

1.6 Concepto de componente gráfico

Los componentes gráficos son cada uno de los elementos que se pueden añadir a la vista actual o a un contenedor —componente gráfico que puede contener otros componentes—, que, generalmente, tiene una representación visual y con el cuál, el usuario interactúa. Estos componentes tienen un comportamiento inherente, que consiste en una respuesta gráfica cuando son interactuados —generalmente una animación— y un comportamiento adquirido, que consiste en la ejecución de un manipulador de evento, creado por el desarrollador, cuando se requiere un tipo de respuesta a cierta interacción —eventos, ver sección 1.11—. Todos los componentes derivan de la clase base *Drawable*, una clase abstracta, donde se declaran métodos que van a utilizar todos los componentes gráficos, la mayoría virtuales, y gran parte de ellos no tienen definición. Estos métodos son tales como los relacionados con la posición del componente dentro de su contenedor —vista o contenedor—, como son las funciones *getX* o *setWidth* —para obtener su posición horizontal y poner su ancho, respectivamente—, los encargados de gestionar eventos, como *handleClickEvent* —manipulador de evento *Click* o pulsación seguida de liberación—, *handleTickEvent* —manipulador del evento *Tick* o temporización—, o métodos relacionados con la representación gráfica como son *draw*, que determina qué se pinta y dónde se pinta, o *getSolidRect* que establece el área mayor a ser redibujada en caso necesario —ver sección 1.13.3—. Como atributos, tiene los relacionados con características visuales, como *visible* y *touchable* que determinan si el control está visible y puede ser tocado, respectivamente, con características de pertenecía, como son *parent* y *nextSibling*, que determinan el objeto que lo contienen o el siguiente objeto en la cadena de pertenencia, respectivamente, o características dimensionales como es *rect*, que representa el área rectangular que contiene al componente.

La clase base de todos los controles gráficos —*Drawable*— tiene dos especializaciones representadas en las clases derivadas *Container* y *Widget* [1]:

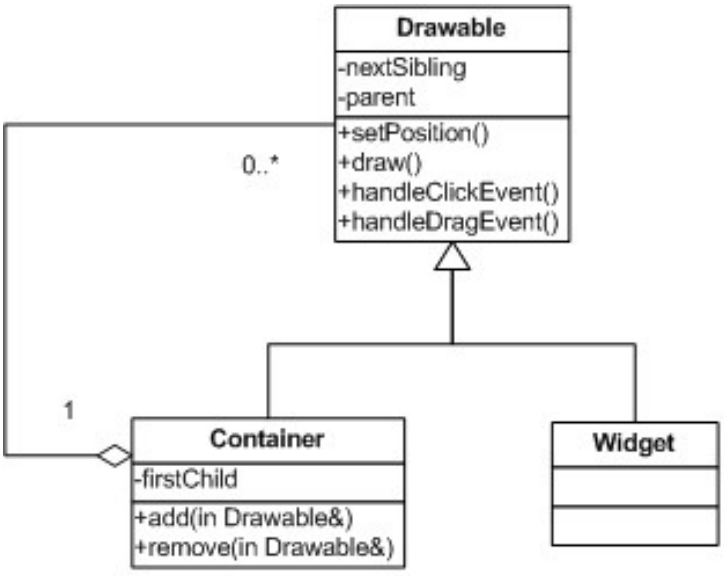


Ilustración 1.17: Jerarquía de clases de los componentes gráficos

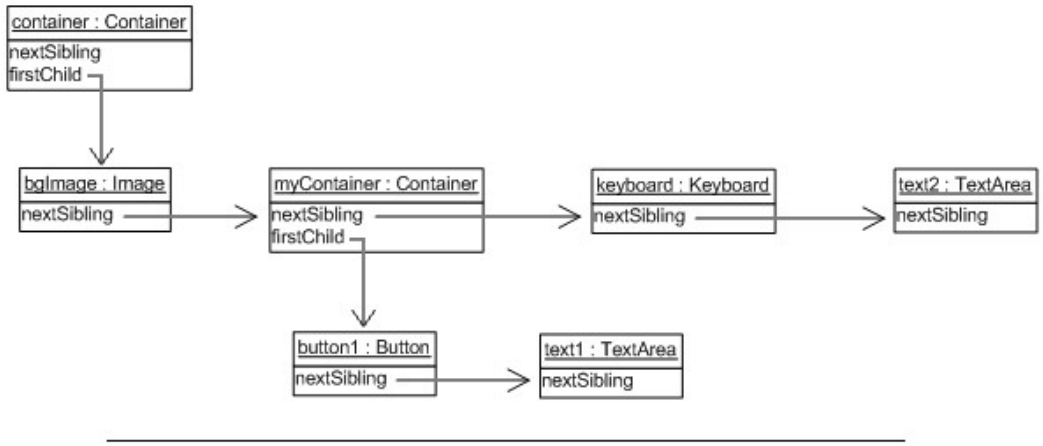
La clase *Widget* sigue siendo una clase abstracta —ya que no implementa todos los métodos de la clase *Drawable*—, lo que implica que no se puedan crear objetos directamente de esta clase. Pero es la clase base para todos los componentes gráficos que pueden ser añadidos a la vista o a un contenedor, y ellos mismos no pueden contener otros componentes, es decir, que no son contenedores. La clase *Widget* define los métodos necesarios,

declarados en *Drawable*, para que sus derivados puedan ser añadidos a un contenedor, estos métodos son: *getLastChild* y *getInvalidatedWidgets*. El método *getLastChild*, es llamado por la vista —o contenedor—, enviando al componente contenido, como argumento de este método, las coordenadas en las que se ha producido un evento —coordenadas que están dentro del componente contenido—, y un puntero donde el componente contenido debe poner la dirección del último componente que él mismo contenga y que esté en las coordenadas pasadas. Al tratarse de la clase *Widget*, no contiene más componentes que él mismo, así que rellena el puntero pasado con su propia dirección. Este relleno de dirección solo se produce si el componente es visible y tocable —tiene las propiedades de *visible* y *touchable* a verdadero—. Por otro lado, el método *getInvalidatedWidgets*, es llamado —por parte de la vista o contenedor— en todos aquellos componentes que han de ser redibujados. Sus argumentos son el área rectangular donde se debe producir el redibujado —que está dentro del componente solicitado— y un vector donde se han de añadir todos aquellos componentes, contenidos en el componente solicitado, que estén sobre el área pasada. Como se trata de la clase *Widget*, el único componente que contiene es él mismo, y por tanto se añade al vector pasado. La implementación de estos dos métodos en la clase *Widget* parece no tener mucho sentido, ya que ambos preguntan por posibles componentes añadidos al componente preguntado, y este, al ser Widget, no puede contener otros componentes más que a sí mismo. Esto es debido a TouchGFX realiza estás «preguntas» a todos los componentes de la vista, independientemente si son contenedores o no, lo que obliga a que los componentes que no sean contenedores a responderlas con su propia dirección o añadirse a sí mismos al vector pasado. Este método de actuar —no teniendo en cuenta si el componente es un contenedor o no—, facilita enormemente el algoritmo de redibujado.

La clase *Container*, en cambio, puede contener a otros componentes, y en este caso sí tiene sentido la implementación de los dos métodos anteriores. En el caso del primero, *getLastChild*, el contenedor buscará, entre sus componentes contenidos, el último añadido cuya área albergue las coordenadas pasadas como argumento. Se busca el último añadido, ya que, gráficamente, es el que está encima del resto de los componentes contenidos —esto se conoce como mayor orden «z»—. Este método tiene como cometido identificar el componente que tiene que recibir el evento —como pueda ser una pulsación donde estén solapados varios controles y el único que ha de gestionar el evento es el que se encuentra encima del todo—. En el caso del segundo método, *getInvalidatedWidgets*, el contenedor deberá añadir al vector, pasado como argumento, todos los componentes contenidos que coincidan, en totalidad o en parte, con la región rectangular pasada como argumento. Desafortunadamente, el cómo lo hacen estas dos funciones dentro del contenedor es desconocido, ya que su código está compilado dentro de la librería facilitada en la versión de evaluación. De todas formas, el funcionamiento de estos métodos es de uso interno en la actividad de *TouchGFX* y transparentes para el desarrollador, a no ser que se desee crear una nueva clase de contenedor —y en contadas ocasiones se necesitará implementar estas funciones al crear un nuevo contendor—. Otros métodos de interés, que sí serán utilizados por el desarrollador son, principalmente, el de añadir un componente al contenedor, método *add*, donde se pasa el componente a añadir como referencia, y el método *remove*, que quita del contenedor el componente especificado en el argumento de esta función. Otros métodos útiles son *contains*, que determina si el componente pasado como argumento está contenido dentro del contenedor o *insert*, que inserta el componente deseado después del especificado y ya contenido en el contenedor. La clase *Container*, al contrario que la clase *Widget*, no es una clase abstracta y puede crear objetos.

Así pues, tal y como se muestra en la figura anterior, los componentes gráficos —*Drawable*—, pueden ser componentes simples —descendientes de *Widget*—, o componentes compuestos o contenedores —*Container* o descendientes de este—. En la figura se observa que un contenedor puede tener cero o varios componentes gráficos — *Drawable*—, mientras que un componente gráfico solo puede tener un contenedor —que puede ser, propiamente un contenedor o la vista—.

Como atributo de la clase *Container*, y solo para uso interno, ya que es protegido, tiene el puntero *firstChild*, que apunta al primer componente contenido. A partir de este puntero se puede acceder al resto de componentes mediante el puntero *nextSibling*, que todo *Drawable* posee, a modo de lista enlazada [1]:




```
1) container.add(bgImage)
2) container.add(myContainer)
3) container.add(keyboard)
4) container.add(text2)
5) myContainer.add(button1)
6) myContainer.add(text1)
```

Ilustración 1.18: Recorrido de componentes en un contenedor

En esta figura —sacada del manual de *TouchGFX*—, se muestra un contenedor, llamado *container* —en la parte superior izquierda—, que contiene varios componentes, entre ellos otro contenedor, llamado *myContainer*. El contenedor más general —el que contiene todo— contiene tres componentes simples que representan la imagen de fondo del contenedor—*bgImage*—, un componente teclado —*keyboard*— y un área de texto —*text2*—. Entre la imagen de fondo y el teclado, está insertado el otro contenedor. El componente que aparece más abajo gráficamente en pantalla es el primer añadido, en este caso la imagen de fondo, mientras que el que se encuentra arriba del todo es el último añadido, en este caso el área de texto *text2*. El contenedor *myContainer* contiene otros dos componentes, pero siempre estarán por debajo del componente que representa el teclado, ya que *myContainer* está debajo de él. Este contenedor contiene un botón —*button1*— y un área de texto —*text1*— , quedando el texto por encima del botón ya que el texto fue añadido después, pero en cualquier caso, ambos están debajo del teclado. Para conseguir este esquema, debajo de la figura se muestra el orden en el que hay que añadir los componentes a los distintos contenedores. Los puntos del 1 al 4 han de estar en este orden, y los puntos 5 y 6 también. Pero podrían haberse añadido primero los componentes del contenedor *myContainer* y luego los del contenedor principal —incluyendo la agregación de *myContainer* en él—, y el resultado habría sido el mismo. Es decir, siguiendo la numeración representada en la figura, este orden podría haber sido: 5-6-1-2-3-4.

En esta figura también se muestra como se relacionan los componentes de un contenedor a través de los punteros *firstChild* y *nextSibling*. Como se comentó, *firstChild* apunta al primer componente añadido al contenedor —el componente que está más abajo o de orden «z» menor—. Se trata de un puntero que solo lo tienen los contenedores. Sin embargo *nextSibling* lo tienen todos los componentes —sean o no contenedores— para que se realice el enlace entre componentes contenidos en un mismo contenedor. Así, el contenedor principal, *container*, tiene ambos punteros, pero solo *firstChild* tiene una dirección válida ya que este contenedor no está contenido dentro de otro y por tanto, no tiene que estar enlazado al siguiente componente. Si, por ejemplo, el contenedor principal quisiera acceder al componente text1 del contenedor *myContainer*, el código sería el siguiente:

```
1 Drawable *componente = firstChild->nextSibling->firstChild->nextSibling;// se obtiene text1
```

Código 1.1: Acceso interno a elementos de un contenedor

Este código se supone que es interno al contenedor —perteneciente algún método como pueda ser *contains*—, ya que los punteros son miembros protegidos y no se tiene acceso a ellos desde fuera de la clase. Para llegar hasta *text1*, primero se desreferencia el puntero *firstChild* del contenedor principal, que apunta a la imagen de fondo, luego se desreferencia el puntero *nextSibling* de esta imagen, que apunta al contenedor *myContainer*, luego se desreferencia su puntero *firstChild* que apunta a su primer elemento contenido, *button1*, y finalmente se desreferencia su puntero nextSibling, que apunta al componente buscado *text1*. Ya que lo que maneja el contenedor son punteros de tipo *Drawable*, será necesario realizar una conversión explícita a la clase descendiente correcta para poder operar con el componente en cuestión, en este caso se debería hacer:

```
1 Text *texto = (Text*) componente;
2 // o también, en una sintaxis más de c++:
3 Text *texto = static_cast< Text*>(componente);
```

Código 1.2: Conversión explícita desde el tipo Drawable

1.7 Uso de imágenes en los componentes gráficos

Existen varios controles gráficos en *TouchGFX* que pueden albergar imágenes. Todos ellos disponen de un método que admite como argumento un objeto de tipo *Bitmap*, que representa la imagen. Este objeto, a su vez, necesita en su constructor un identificador que indexe el array que representa la base de datos de las imágenes, que junto con los arrays que contienen la información gráfica de los puntos de cada imagen, se creados al compilar los recursos —ver sección 1.4—. Por ello, en todos los archivos fuentes de las vistas, se necesita incluir la cabecera *BitmapDatabase.hpp*, que contiene los identificadores de cada imagen para poder indexar el array de imágenes. Cada nombre de los identificadores de este archivo, se corresponde con el nombre original del archivo gráfico antes de compilar —expresado en mayúsculas—, precedido por el prefijo «*BITMAP_*» y terminado con el sufijo «*_ID*». El valor numérico de cada uno de estos identificadores se corresponde con el índice del array imágenes para indexar la imagen correcta:

```
1  //En archivo de cabecera de la vista, ejemplo FftView.hpp
2  class FftView : public View<FftPresenter>
3  {
4    public:
5      . . .
6      virtual void setupScreen();
7      . . .
8    private:
9      . . .
10     Image rejillaAnterior;
11     . . .
12 }
13
14 //En archivo fuente de la vista, ejemplo FftView.cpp
15 void FftView::setupScreen()
16 {
17     . . .
18     rejillaAnterior.setBitmap(Bitmap(BITMAP_REGILLA_ID));
19     rejillaAnterior.setAlpha(254);
20     . . .
21     add(rejillaAnterior);
22 }
```

Código 1.3: Utilización de imágenes en TouchGFX

En esta porción de código, se muestra que inicialmente se debe declarar el objeto en la parte privada de la vista —línea 10—. Luego, se configura dentro del método *setpScreen* de la misma. Primero se le ha de asignar el objeto de tipo *Bitmap*. Para ello, en el argumento del método de asignación de imagen al objeto —método *setBitmap*—, se crea un objeto de tipo *Bitmap*, mediante la llamada al constructor de esta clase, pasándole como argumento el identificador de la imagen que será usada, de forma interna, para indexar la imagen dentro del array de la base de datos de imágenes.

Luego se realizan más configuraciones sobre este objeto, como la transparencia —línea 19— o la posición —no especificada y por tanto su posición es (0,0)—. Finalmente se ha de añadir a la vista —línea 21— para que pueda ser visible y manejable en el entorno gráfico.

También es posible asignar un rango de imágenes a un objeto gráfico. Existe el control *AnimatedImage* que espera una serie de imágenes y crear con ellas una animación al pasar de una a otra de forma periódica. Para especificar un rango de imágenes, se ha de definir al imagen inicial y la final; las imágenes intermedias deben tener identificadores consecutivos —en su identificador aparece un número correlativo—, y para ello, los nombres de los archivos de imágenes originales antes de compilar, deben tener esos nombres. Por ejemplo, si se crea un objeto llamado *animacion*, de este tipo y se ejecuta su método *animation.setBitmaps(BITMAP_ANI_01_ID, BITMAP_ANI_14_ID)*, el objeto está esperando que existan doce imágenes más en la base de datos de imágenes y que sus identificadores sean del tipo *BITMAP_ANI_XX_ID*, donde *XX* son todos los números del 2 al 13. .

1.8 Uso de textos en los componentes gráficos

En TouchGFX existe la clase *TypedText* que encapsula, conjuntamente, la cadena de texto y su tipografía —ver sección 1.4.3—. En su constructor admite como argumento el índice de la cadena dentro del array de cadenas tratadas en el entorno gráfico. Como se pretende que todo el proceso sea lo más transparente posible al desarrollador, se provee una serie de constantes cuyo nombre hace referencia a la cadena a utilizar y cuyo valor es el del índice en el array de cadenas. Estas constantes son creadas al compilar los recursos y almacenadas en el archivo de cabecera *TextKeysAndLanguages.hpp* —ver sección 1.4—, con lo que debe estar incluido en todos los archivos fuentes de todas las vistas. Cualquier control gráfico que requiera una cadena, dispondrá de un método que requiera un objeto de tipo *TypedText*. Así, cualquier control gráfico que requiera una cadena, seguirá estos pasos en el código:

```
1  //En archivo de cabecera de la vista, ejemplo FftView.hpp
2  class FftView : public View<FftPresenter>
3  {
4    public:
5        . . .
6        virtual void setupScreen();
7        . . .
8    private:
9        . . .
10       TextArea textFrecCentral;
11       . . .
12 }
13
14 //En archivo fuente de la vista, ejemplo FftView.cpp
15 void FftView::setupScreen()
16 {
17     . . .
18     textFrecCentral.setTypedText(TypedText(T_TEXTOFRECCENTRAL));
19     textFrecCentral.resizeToCurrentText();
20     textFrecCentral.setXY(contContrl.getWidth()/2-textFrecCentral.getWidth()/2,5);
21     textFrecCentral.setColor(Color::getColorFrom24BitRGB(0x00, 0xFF, 0x00));
22     . . .
23     add(textFrecCentral);
24 }
```

Código 1.4: Utilización de una cadena constante en TouchGFX

Primero se ha de declarar el control que va a utilizar la cadena de texto en la parte privada de la vista —como cualquier otro control—. Luego, en el método *setupScreen* de la vista se configura el control. El paso a destacar es el que se encuentra en la línea 18. El control utiliza su método *setTipedText* para agregar la cadena de texto con identificador *T_TEXTOFRECCENTRAL* —que corresponde a la cadena «*TextoFrecCentral*» del archivo de cadenas *texts.xlsx*, ver sección 1.4—. Para ello utiliza la clase *TypedText*. Es decir, que el objeto *textFrecCentral*, utiliza su método *setTipedText*, que espera como argumento un objeto de tipo *TypedText*, y el constructor de este último, el índice que corresponde a la cadena deseada —*T_TEXTOFRECCENTRAL*—. Luego se producen diversas configuraciones más sobre el objeto *textFrecCentral*, para finalmente se añadido a la vista —línea 23—.

El uso de una cadena variable es similar pero con alguna diferencia:

```
1  //En archivo de cabecera de la vista, ejemplo FftView.hpp
2  class FftView : public View<FftPresenter>
3  {
4    public:
5      . . .
6      virtual void setupScreen();
7      . . .
8    private:
9      . . .
10     TextAreaWithOneWildcard texPanFrecCent;
11     Unicode::UnicodeChar cadenaPanFrecCent[25];
12     . . .
13 }
14
15 //En archivo fuente de la vista, ejemplo FftView.cpp
16 void FftView::setupScreen()
17 {
18    . . .
19     texPanFrecCent.setWildcard(cadenaPanFrecCent);
20     texPanFrecCent.setTypedText(TypedText(T_PANTALLAFRECCENTRAL));
21     texPanFrecCent.setColor(Color::getColorFrom24BitRGB(0,250,200));
22     Unicode::strncpy(cadenaPanFrecCent, "843,750 kHz", strlen("843,750 kHz")+2);
23     texPanFrecCent.resizeToCurrentText();
24     . . .
25     add(textFrecCentral);
26 }
```

Código 1.5: Utilización de una cadena variable en TouchGFX

Al igual que en el caso anterior, se declara el control que usará la cadena —línea 10—, pero en este caso variable. Por ello necesita la ayuda de otro objeto, que es de tipo Unicode, que representa una cadena Unicode —línea 11—. En la configuración del objeto, dentro del método *setupScreen* de la vista, se asigna al objeto de usará la cadena variable, el objeto auxiliar de tipo Unicode —línea 19—. A partir de aquí, su configuración es idéntica a la de la cadena constante. Para variar el contenido de la cadena, basta con rellenar el objeto de tipo Unicode asociado para que el objeto que utiliza la cadena variable —*texPanFrecCent*— refleje el cambio —línea 22—. Como se aprecia en el código —línea 20— también se debe asignar al objeto que utiliza la cadena variable, un objeto *TypedText* con el identificador de la cadena del archivo texts.xlsx —que será un comodín de tipo <*>, ver sección 1.4—. Esto es debido a que estas cadenas —las definidas en el archivo texts.xlsx—, aparte de contener la cadena de texto —en este caso es una cadena comodín—, contienen información de la fuente y el tamaño.

1.9 Patrón de programación Modelo – Vista – Presentador (MVC) en TouchGFX

El patrón MVC, nombrado como MVP en *TouchGFX* —Modelo, Vista y Presentador—, es un patrón que intenta separar la lógica de la aplicación de la correspondiente representación gráfica [22]. De esta forma, una misma aplicación puede mostrarse gráficamente de diversas maneras, en vistas diferentes, o una aplicación compuesta de varias aplicaciones, puede asignársele una vista diferente a cada una de ellas. Este patrón consta de tres elementos: una Vista, un Controlador o Presentador y un Modelo. La Vista es la interfaz con la que el usuario interactúa. Contiene controles gráficos que el usuario manipula produciendo órdenes que debe tratar la Vista. Así mismo, la Vista recibe las órdenes procedentes del Presentador como consecuencia de cambios en el Modelo. El Presentador recibe las órdenes del usuario a través de la Vista y las trata, normalmente, mediante manipuladores de eventos. En ellos se pueden modificar las propiedades de la Vista o el Modelo, lo que puede provocar, en este último caso, que se disparen órdenes desde el Modelo al Presentador y de allí a la Vista. La

función principal del modelo es la de mantener el estado de las vistas entre cambios de las mismas. Sin embargo estos modos de actuación, de cada una de las entidades de este patrón, no son tan rígidos. En la sección 1.11.4, dedicada a eventos entre estos elementos, se tratarán diversos escenarios de comunicación entre ellos.

Las clases encargadas de conformar este patrón en *TouchGFX,* así como las clases relacionadas, se muestran en el siguiente diagrama:

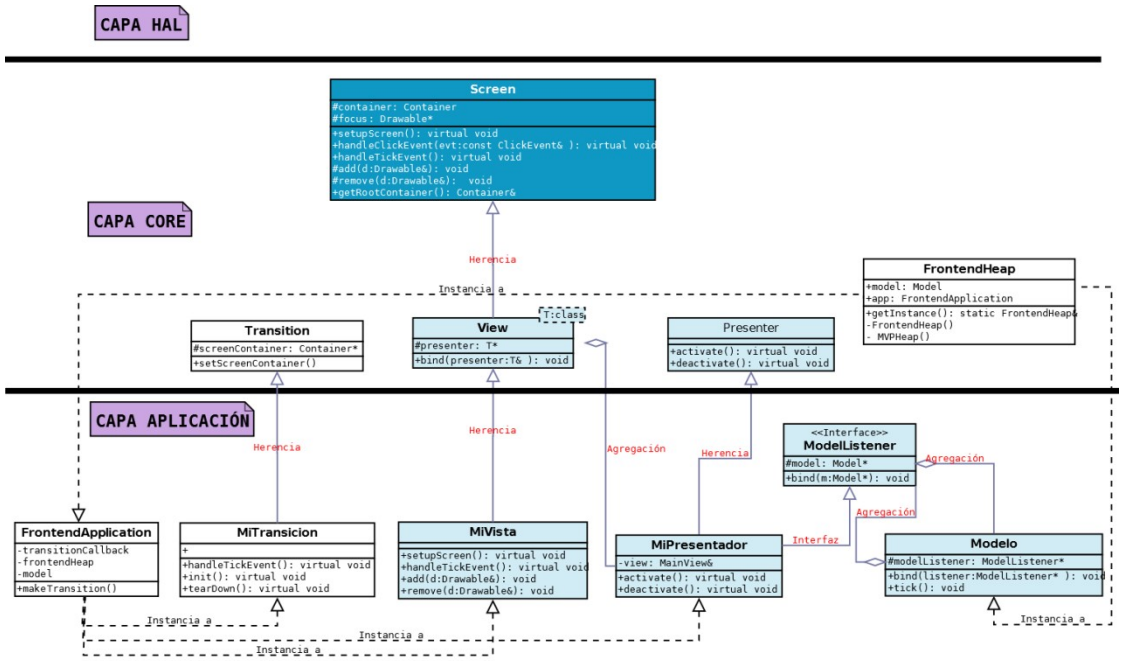


Ilustración 1.19: Relación de clases del patrón MVC en TouchGFX

El patrón está constituido por las clases marcadas en azul localizadas en la capa de aplicación, en este caso, son: *MiVista*, *MiPresentador*, *Modelo* y el interfaz *ModelListener*. *MiVista* es la especialización, hecha por el programador, de la clase *View* de la capa *Core*, que a su vez, desciende de la clase *Screen*. La clase *View* dispone de un puntero de tipo *Presenter*, que será cargado con la dirección del presentador utilizado en la capa de aplicación, en este caso, *MiPresentador*. De esta forma, y a través del mecanismo de polimorfismo, la clase *MiVista* accede a *MiPresentador* a través de este puntero. El presentador utilizado en la capa de aplicación, *MiPresentador*, desciende de la clase *Presenter* en la capa *Core*, pero también desciende de la interfaz[4] *ModelListener*. Mediante esta segunda herencia, *MiPresentador* es capaz de acceder al modelo a través del puntero *model* que *ModelListener* posee. Pero esta no es la función de *ModelListener*. Al ser una de las clases base de *MiPresentador*, puede declarar funciones virtuales con cuerpo vacio para que este las redefina. El Modelo tiene un puntero a *ModelListener*, lo que le permite llamar a estas funciones virtuales definidas en él —*ModelListener*—. Así, si estas funciones solo están definidas en *ModelListener* con cuerpo vacio, no sucederá nada; pero si estas funciones están redefinidas en *MiPresentador*, serán las llamadas, debido al mecanismo de polimorfismo. Esto es posible ya que, en la carga inicial del par Vista-Presentador, o en el cambio de este par al producirse el cambio entre pantallas, el puntero que contiene el Modelo, que apunta a *ModelListener*, es cargado con la dirección del presentador, en este caso *MiPresentador*. Esto posibilita que en *ModelListener* se definan, con cuerpo vacio, todas las funciones que el Modelo va a utilizar referente a todas las posibles vistas que se van a manejar. Así, si se tiene una pantalla concreta cargada —el término pantalla hace referencia al par Vista-Presentador en uso—, en el presentador solo se redefinirán aquellas funciones definidas en *ModelListener* que estén relacionadas con la pantalla cargada. De esta forma, al ser llamadas estas funciones por parte del Modelo,

[4] En C++ no existe la entidad interfaz —*interface*— como tal, al contrario cómo sucede en otros lenguajes como JAVA. Realmente se trata de herencia múltiple, aunque en este caso, funcionalmente, se adapta al patrón de programación *Interface* [22].

se ejecutarán las redefinidas en el presentador de la pantalla en uso, mientras que si llama a otras funciones relacionadas con otras pantallas, se ejecutarán las versiones con cuerpo vacio de *ModelListener*. Por otro lado, *MiPresentador* posee una referencia para poder acceder a la vista en uso —*MiVista*—. *MiTransicion* es una clase dedicada a realizar el cambio gráfico entre diferentes pantallas —pares Vista-Presentador—. La clase dedicada a la reserva estática de memoria para la terna vista-presentador-transición más grande es *FrontendHeap*. Esta clase reserva, en tiempo de compilación, tres aéreas de memoria para albergar a la vista, el presentador y la transición. También instancia al Modelo y al objeto de tipo *FrontendApplication* encargado, entre diversas funciones, de crear y enlazar los objetos concretos de la pantalla en uso —el par vista-presentador más la transición—.

En definitiva, el modelo tiene acceso al presentador; el presentador tiene acceso al modelo y a la vista y esta última, lo tiene al presentador. Sin embargo, tal y como está implementado el patrón, la vista no tiene acceso al modelo, aunque este será un punto de desarrollo que será tratado en el capítulo cuatro.

El código que se alojará en el modelo consistirá en todas aquellas propiedades de controles de la vista que se considere que deben ser mantenidas entre cambios de pantallas. Estas propiedades han de ir en la parte privada, y para el acceso a ellas —escritura y lectura—, se necesitan los métodos necesarios que se declararán en la zona pública:

```
1  class ModelListener;
2  class Modelo
3  {
4  public:
5      Modelo();
6      void ponerAtributoModelo(int valor);
7      int obtenerAtributoModelo(void);
8      . . .
9      //Demás funciones de acceso a los atributos del modelo
10     void bind(ModelListener* listener) { modelListener = listener; }
11     void tick();
12
13 private:
14     int AtributoModelo;
15     . . .
16     //Demás atributos del modelo
17 };
```

Código 1.6: Código esqueleto de la cabecera del modelo

Además, como métodos importantes del modelo están *bind*, encargado de establecer la conexión entre el modelo y *ModelListener* —llamado por *FrontenApplication*, en el cambio de pantallas— y *tick*, encargado de realizar tareas periódicas. Este último método es llamado en cada actualización de la pantalla —idealmente en cada refresco vertical—, y, como se mostrará en el capítulo cuatro, será el lugar de conexión entre el entorno gráfico y la aplicación de usuario —a través del sistema operativo de tiempo real—. En el cuerpo de esta función se producirán las llamadas a las funciones definidas en *ModelListener* y redefinidas en el presentador. En el siguiente código se muestra la llamada a la función *actualizaMiVista* de *ModelListener*, realizada desde la función *tick* del modelo:

```
1  void Modelo::tick()
2  {
3   if(modelListener)
4   {
5    //acciones repetitivas en cada evento de actualización de pantalla (tick)
6    . . .
7    modelListener->actualizaMiVista(/*argumento para actualizar la vista*/);
8   }
9  }
```

Código 1.7: Llamada a las funciones de ModelListener desde el método tick del modelo

En *ModelListener* se encuentra el puntero al modelo —para que el presentador pueda acceder al modelo—, y la función *bind* para cargar este puntero —que lo hará *FrontenApplication* en el cambio de pantallas—. En la zona pública se declaran y definen, con cuerpo vacio, todas las funciones que serán redefinidas en algún presentador de los constituyentes en la aplicación gráfica. En el ejemplo de código siguiente, se define el método *actualizaMiVista*, que deberá ser redefinida en el presentador *MiPresentador*:

```
1  class ModelListener
2  {
3    public:
4       ModelListener() : modelo(0) {}
5       void bind(Model* m) {modelo = m;}
6       virtual void actualizaMiVista(/*argumento para actualizar la vista*/){}
7       //demás funciones para comunicación con el resto de vistas
8    protected:
9       Model* model;
10 };
```

Código 1.8: Funciones de ModelListener para comunicación con el presentador

En el presentador —*MiPresentador*—, se redefine esta función, que generalmente consistirá en la llamada a una función de la vista cuyo nombre suele ser el mismo:

```
1  class MiVista;
2
3  class MiPresentador : public Presenter, public ModelListener
4  {
5    public:
6      MiPresentador(MiVista& v);
7      virtual void activate();
8      virtual void deactivate();
9
10          virtual void actualizaMiVista(/*argumento para actualizar la vista*/)
11                  { view.actualizaMiVista(/*argumento para actualizar la vista*/);}
12          . . .
13          //posibles manipuladores de eventos delegados desde la vista
14
15   private:
16     MiVista& view;
17 };
```

Código 1.9: Código esqueleto de la cabecera del presentador

En la parte privada del presentador existe una referencia a la vista que es cargada por *FrontendApplication*, en el cambio de pantallas, a través del constructor. En la parte pública están las funciones *actívate* y *desactívate*, que son llamadas por *FrontendApplication* y sirven, respectivamente, para la configuración inicial y final en el cambio entre pantallas —en este proyecto, estas dos funciones no ha sido utilizadas—. En esta parte, se hayan las redefiniciones de las funciones de *ModelListener* —solo aquellas que deben ser tratadas en el par vista-presentador actual—.

En la vista, en su parte privada, se declaran todos los controles gráficos a ser utilizados, mientras que la configuración de los mismos se realiza en el método *setupScreen* localizado en la parte pública. En esta misma zona se localiza el método *tearDownScreen*, llamado por *FrontentApplication* al iniciarse el cambio de pantalla —

no utilizado en este proyecto—, las funciones llamadas por el presentador, y todas aquellas funciones de utilidad de las quiera disponer el programador. El esquelto de la clase de la vista es el siguiente:

```
class MiVista : public View<MiPresentador>
{
    public:
        MiVista();
         virtual void tearDownScreen()
         virtual void setupScreen();
            {//configuración de los controles
              //declarados en la parte privada
            }
         void actualizaMiVista(/*argumento para actualizar la vista*/)
             {//actualizar los controles con los argumentos;}
         . . .
    private:
            //declaración de controles gráficos
};
```

Código 1.10: Esqueleto del código de la cabecera de la vista

1.10 Reserva estática de memoria del Modelo – Vista – Presentador

TouchGFX usa un modelo estático de memoria RAM, donde reserva la cantidad necesaria para albergar los objetos que representan el patrón MVC. Hacerlo de forma estática significa que se debe conocer la cantidad de memoria para el patrón en tiempo de compilación. El modelo es un elemento que no cambia y siempre está presente a lo largo de la ejecución. Determinar la cantidad de memoria necesaria para él no es un problema —mediante *sizeof(Model)*—, sin embargo, solo hay una pareja vista-presentador activa de las disponibles en la aplicación gráfica. Por ello, la cantidad de memoria a reservar es aquella para albergar a la vista y presentador que sean más grandes. De igual forma, y aunque no pertenezca al patrón, se ha de reservar memoria para el objeto transición, que es el encargado de realizar, visualmente, el cambio entre pantallas —este objeto siempre es necesario aunque este cambio visual consista en no hacer nada, simplemente que aparezca la nueva pantalla de golpe—. La clase donde se albergan, físicamente, todos estos elementos, es la clase *FrontedHeap*:

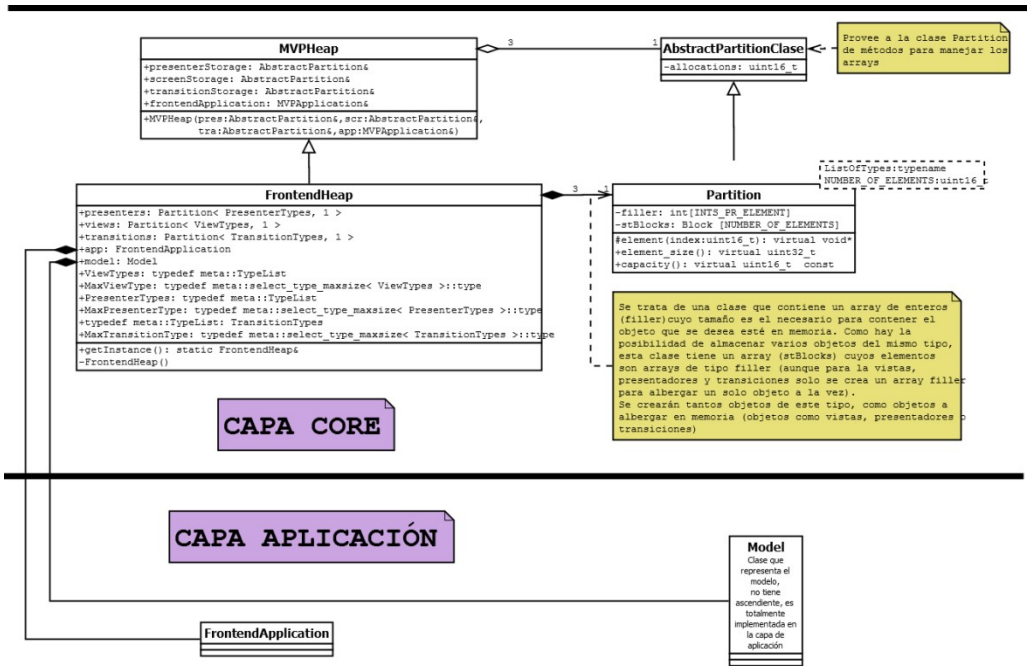


Ilustración 1.20: Relación de clases entorno a FrontedHead

Para poder hacer su trabajo, *FrontedHeap*, declara tres objetos de tipo *Partition*, una clase que contiene un array de enteros —*filler*— que sirve para alojar un objeto en memoria. La clase *Partition* también alberga una array —*stBlocks*—, que puede contener varios arrays de tipo *filler*, especificando este dato al crear el objeto *Partition* —esto es posible ya que se le pasa este dado a la plantilla *Partition*—.

La declaración de los objetos *Partition* en *FrontedHeap* es la siguiente:

```
1  class FrontendHeap : public MVPHeap
2  {
3  public:
4      . . .
5      Partition< PresenterTypes, 1 > presenters;
6      Partition< ViewTypes, 1 > views;
7      Partition< TransitionTypes, 1 > transitions;
8      FrontendApplication app;
9      Model model;
10 private:
11     . . .
12 };
```

Código 1.11: Declaración de los alojamientos del patrón MVC

En las líneas 5, 6 y 7 se definen los alojamientos de tipo *Partition* para el mayor presentador, la mayor vista y la mayor transición, respectivamente, que se alojaran en la aplicación gráfica. Cada uno de estos objetos *Partition*, esperan, como argumentos —argumentos de plantilla— un lista que contiene todos los objetos del tipo deseado a alojar —presentadores, vistas o transiciones— y cuantos alojamientos para ese tipo se desea reservar. En este caso, se crean los tres objetos *Partition* para albergar, en cada uno de ellos, un presentador —*presenters*—, una vista —*views*— y una transición —*transitions*— cuyo tamaño, para cada elemento, será el mayor de los que se utilice en la aplicación gráfica —primer argumento de plantilla de *Partition*—. Solo se alojará un elemento en cada objeto *Partition* —segundo argumento de plantilla—. La clave para determinar el mayor tamaño de cada uno, es mediante la lista pasada como primer argumento de plantilla a cada objeto *Partition*. Esta lista será la única cosa de la que se tendrá de preocupar el desarrollador y que será mostrado al final de esta sección.

En la clase *FrontedHead*, se crean los objetos de tipo *FrontedApplication* —línea 8—, que representa la aplicación gráfica que controla el cambio de pares vista-presentador –ver sección 1.12—, y el objeto que representa el modelo —línea 9—. Hay que aclarar que en *FrontedHeap* no se crean los objetos vista, presentador y transición actualmente en uso, sino que crean las áreas de memoria donde serán albergados. Es *FrontedApplication* quien crea estos objetos albergándolos en los objetos *Partition* de *FrontedHeap* —ver esquema UML de la sección 1.9—.

Todo el proceso se alojamiento de memoria es transparente al desarrollador, desde su punto de vista, la labor que debe realizar es especificar una serie de listas —unas plantillas facilitadas por *TouchGFX*— en donde se especifique, en cada una de ella, las vistas, los presentadores y las transiciones utilizadas. Esto se hace dentro de la parte pública de la clase *FrontedHeap* con la siguiente sintaxis:

```
1  class FrontendHeap : public MVPHeap
2  {
3    public:
4     typedef meta::TypeList< FftView,
5                             meta::Nil
6                           > ViewTypes;
7
8
9      typedef meta::TypeList< FftPresenter,
10                            meta::Nil
11                          > PresenterTypes;
12
13
14     typedef meta::TypeList< NoTransition,
15                            meta::Nil
16                          > TransitionTypes;
17
18     . . .
19   private:
20    . . .
21 };
```

Código 1.12: Determinación del mayor tamaño de cada elemento MVC

Se han de definir tres listas de tipo *meta::TypeList* —líneas 4, 9 y 14—, una para las vistas, otra para los presentadores y otra para las transiciones. Cada elemento de estas listas serán las clases de las vistas, presentadores y transiciones utilizados en la aplicación gráfica. Los elementos se especifican dentro del identificador de plantilla —«<» y «>»— separados por comas y finalizando la lista con un elemento especial : *meta::Nil*. En el caso del código mostrado —que se corresponde con el código utilizado en este proyecto—, solo hay un elemento de cada lista, ya que solo hay una vista, un presentador y una transición —esta última no tiene funcionalidad pero *TouchGFX* necesita una por cada par vista-presentador—. Para obtener el tamaño en RAM de todo el interfaz gráfico, basta con ejecutar *sizeof(FrontedHeap)*.

1.11 Eventos en TouchGFX

En *TouchGFX* hay varios niveles de eventos. Están los propios de *TouchGFX*, esto es, los generados en su capa HAL y redirigidos al componente responsable de su disparo; la respuesta que el usuario facilita a tal evento mediante métodos en la vista, denominados manipuladores de eventos, que son enlazados a la vista mediante los objetos de tipo *Callback*. Como eventos particulares de *TouchGFX* están los temporizadores, que ejecutan código de forma repetida. Luego están los eventos entre entidades de *TouchGFX*, eventos de más alto nivel que los anteriores. Concretamente este tipo de eventos se tratan entre el Modelo, la Vista y el Presentador. Aunque estos eventos siguen perteneciendo a los tratados por *TouchGFX*, son más propios de patrón MVC. Finalmente están los eventos que se dan entre *TouchGFX* y la aplicación final de usuario o *Backend* —la aplicación realmente útil—. Este tipo de eventos son los que están más alejados de *TouchGFX* pero en contacto con este. De hecho, el fabricante del entorno gráfico no especifica gran cosa acerca de ellos en su manual. Esta es la causa de que sea uno de los objetos de desarrollo de este proyecto.

1.11.1 Eventos desde la capa HAL hacia los componentes gráficos.

Los eventos tales como la pulsación sobre elementos gráficos, el arrate de los mismos, movimientos gestuales como desplazamiento o indicaciones de marcas de temporización, son generados en la capa HAL mientras el controlador de pantalla está barriendo el buffer de primer plano —el buffer en el que no trabaja directamente *TouchGFX* para dibujar; lo hace sobre el secundario—. La comunicación de eventos entre la capa HAL y la CORE se muestra en la siguiente ilustración:

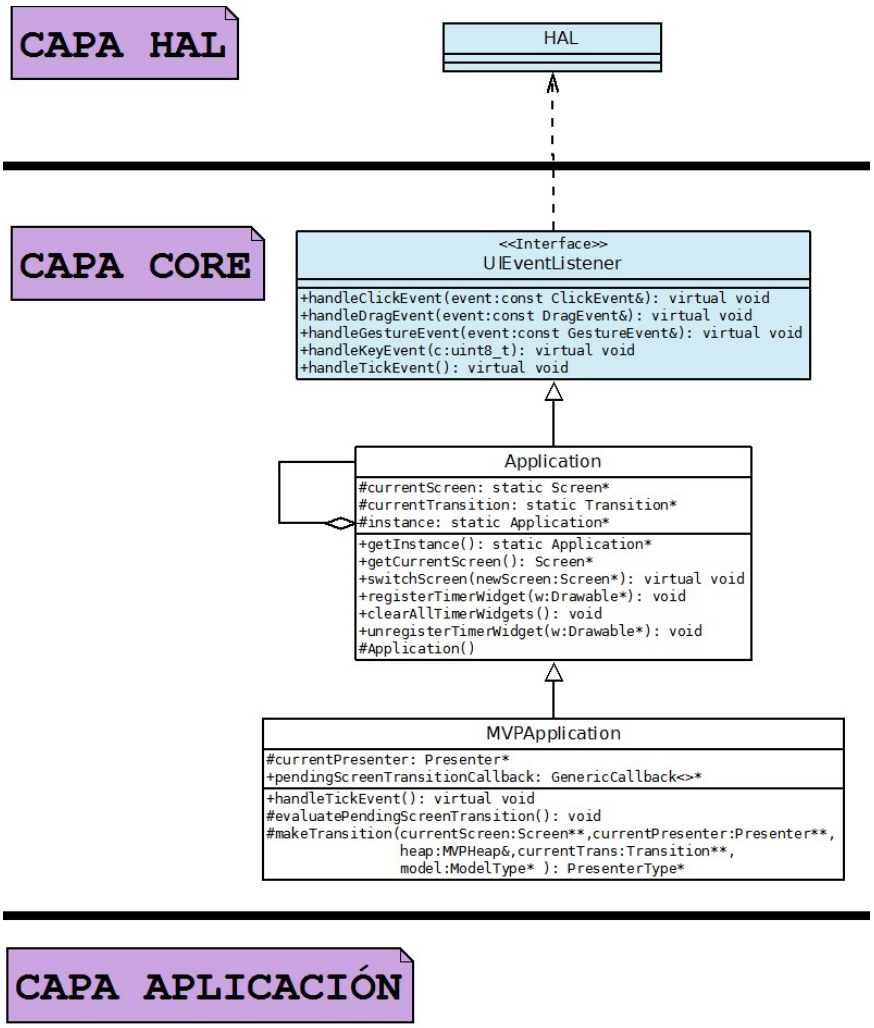


Ilustración 1.21: Esquema UML del interface UIEventListener.

En este gráfico *UML* se muestra la clase *MVPApplication* que representa la aplicación y desciende de la clase virtual *Application*; pero a efectos de comprensión de la transmisión de eventos podemos obviar la existencia de esta última clase. La clase *MVPApplication* va a ser la encargada de crear la vista y el presentador y de transmitir los eventos generados en la capa *HAL* a cada uno de los componentes gráficos. Para ello implementa el interface *UIEventListener*, una clase abstracta cuyos métodos son llamados por la clase *HAL* pasándoles los parámetros correspondientes para poder gestionar dichos eventos. Después, la clase *MVPApplication*, en base a la llamada a estos métodos y a la información pasada a los mismos, transmite el evento al componente gráfico correspondiente.

El código de la clase *UIEventListener* se muestra a continuación [3]:

```
1  class ClickEvent;
2  class DragEvent;
3  class GestureEvent;
4
5  class UIEventListener
6  {
7  public:
8    virtual void handleClickEvent(const ClickEvent& event)
9    {
10   }
11
12   virtual void handleDragEvent(const DragEvent& event)
13   {
14   }
15
16   virtual void handleGestureEvent(const GestureEvent& event)
17   {
18   }
19
20   virtual void handleKeyEvent(uint8_t c)
21   {
22   }
23
24   virtual void handleTickEvent()
25   {
26   }
27
28   virtual ~UIEventListener()
29   {
30   }
31 };
```

Código 1.13: Clase UIEventListener

Todos sus métodos son métodos «vacios», es por ello que la clase *MVPApplication* debe suministrar un código para cada uno de ellos. Realmente la relación entre *MVPApplication* y *UIEventListener* es de herencia, y desde el punto de vista de la capa *HAL*, *MVPApplication* es de la clase *UIEventListener*, es decir, que "la cara" que da *MVPApplication* a la capa *HAL* es como si fuese *UIEventListener*, esto es lo que se conoce como patrón "Interface" [22]. En otros lenguajes este patrón está claramente definido, pero en C++ se consigue mediante herencia múltiple —en este caso es herencia simple—.

Del código anterior cabe resaltar los métodos «*handleClickEvent*», «*handleDragEvent*» y «*handleGestureEvent*», que son los referentes a los eventos que se disparan por actuaciones sobre la pantalla táctil, mientras que «*handleKeyEvent*» es referente a botones físicos en la placa que el usuario deberá implementar por otro medio —no desarrollado en este proyecto— y en cualquier caso el usuario es libre de gestionar los botones físicos sin necesidad de pasar a través de *TouchGFX*. El evento «*handleTickEvent*», debido a su especial relevancia, será tratado más adelanta en el apartado «Temporizaciones».

Los eventos —métodos— que se disparan por actuaciones directas en pantalla, tienen como argumento clases específicas para cada uno de ellos y contienen información concreta de los mismos. Por ello estas clases aparecen como declaraciones adelantadas entre las líneas 1 a 3 del código anterior.

La clase *MVPApplication* «mandará» el evento al componente gráfico que esté en la posición «Z»[5] superior, que cubra la zona de pantalla donde se genera el evento y que esté preparado para ser «tocable»[6] y sea «visible»[6], además traducirá la información de las coordenadas de valores absolutos —teniendo como origen de coordenadas la esquina superior izquierda de la pantalla— a coordenadas relativas al componente que recibe el evento —donde el origen de coordenadas es su esquina superior izquierda—. Para conseguirlo, la clase *MVPApplication* llamará a la función-evento homónima del componente que debe gestionarlo. Por ello, desde el

[5] Orden de solapamiento de los controles gráficos en pantalla, el control de posición Z mayor, se dibuja encima del resto.

[6] Propiedades de los controles gráficos que les permiten recibir eventos de pulsación y gestuales —tocable— y que aparezcan en pantalla —visible—.

punto de vista del programador, cuando desarrolle nuevos componentes gráficos, solo ha de preocuparse de suministrar el método correspondiente al evento a gestionar – «handleClickEvent», para eventos relacionados con pulsaciones sobre el control, «handleDragEvent», para eventos relacionados con el «arrastre» del componente o «handleGestureEvent» para eventos gestuales como pueda ser desplazar el dedo de un sitio a otro dentro del componente –. El desarrollador deberá rellenar estas funciones dotándolas del código necesario para dar servicio al evento concreto. Lo que sí debe conocer el usuario es la estructura de las clases de eventos «*ClickEvent*», «*DragEvent*», y «*GestureEvent*». Estas clases derivan de la clase abstracta «Event» [3]:

```
class Event
{
public:

    typedef enum
    {
        EVENT_CLICK,
        EVENT_DRAG,
        EVENT_GESTURE
    } EventType;

    Event() { }

    virtual EventType getEventType() = 0;

    virtual ~Event() { }
};
```

Código 1.14: Clase Event

Esta clase solo define el tipo enumerado «*EventType*» para determinar qué tipo de evento es el que se ha producido. Las clases de eventos específicos – descendientes de esta – tendrán que implementar el método «*getEvntType*» que devuelva este tipo, y que la clase «*Event*», mostrada en el código anterior, declara como función virtual pura.

Seguidamente se muestra, a modo de ejemplo de las clases de tipo evento, la clase ClickEvent [3]:

```
class ClickEvent : public Event
{
public:
     typedef enum
    {
        PRESSED,
        RELEASED,
        CANCEL
    } ClickEventType;

    ClickEvent(ClickEventType type, int16_t x, int16_t y, int16_t force = 0) :
        _type(type), _x(x), _y(y), _force(force) { }

    virtual ~ClickEvent() { }
    int16_t getX() const { return _x; }
    int16_t getY() const { return _y; }
    void setX(int16_t x) { _x = x; }
    void setY(int16_t y) { _y = y; }
    void setType(ClickEventType type) { _type = type; }
    ClickEventType getType() const { return _type; }
    int16_t getForce() const { return _force; }
    virtual Event::EventType getEventType() { return Event::EVENT_CLICK; }

private:
    ClickEventType _type;
    int16_t _x;
    int16_t _y;
    int16_t _force;
};
```

Código 1.15: Clase ClickEvent

En la línea 22 se define el método antes mencionado, devolviendo como tipo de evento «EVENT_CLICK». En esta clase descendiente se define el tipo *ClickEventType* como un subtipo del evento. El valor de este subtipo es almacenado en el atributo «*_type*» – línea 25 – y es accedido a través del método público *getType* – línea 20 –. Mediante su llamada, el programador podrá concretar qué es lo que ha sucedido. Así, si la función *getType* devuelve el valor de «PRESSED», significa que se ha presionado sobre el componente que recibe el evento; si es «RELEASED», significa que el usuario ha liberado la presión sobre el componente; y si es «CANCEL», que se ha liberado la presión fuera del área del componente. Las coordenadas – relativas – donde sucede el evento, son almacenadas en los atributos «_x» e «_y» y son accedidos mediante los métodos *getX* y *getY* respectivamente. La capa *HAL*, encargada de disparar el evento, rellenará los valores de los atributo del evento mediante los métodos *setX*, *setY*, *setType* y mediante el constructor de la clase.

Cuando se produzca este tipo de evento en un componente creado por el usuario y se implemente el método *handleClickEvent*, la capa HAL lo llamará siempre que se produzca el evento *ClickEventType* sobre él – esta llamada se hace a través de «*MVPApplication*» – pasándole como argumento un objeto de tipo *ClickEvent* relleno con valores concretos. A modo de ejemplo resumido, se muestra el siguiente código:

```
1  class MiComponente: public Widget
2  {
3  public:
4      //partes públicas de la clase: métodos y tipos
5      MiComponente():Widget(),presionado(false) {} //constructor
6  private:
7      //partes privadas de la clase: atributos, métodos privados
8      bool presionado;
9  protected:
10     //partes privadas de la clase pero accesibles desde clases descendientes
11     void handleClickEvent(const ClickEvent & evt)
12     {
13         switch(evt.getType())
14         {
15         case ClickEvent::PRESSED:
16             //código que gestione la presión sobre el componente (de tipo MiComponente)
17             presionado=true;
18             break;
19
20         case ClickEvent::RELEASED:
21             //código que gestione la liberación sobre el componente
22             if(presionada)
23             {
24                 presionado=false;
25                 //se ha pruducido un "click"
26                 //Comprobar que hay evento definido de usuario y si
27                 //es así proceder a llamarlo.
28             }
29             break;
30
31         case ClickEvent::CANCEL:
32             //código que gestione la liberación fuera del componente
33             presionado=false;
34             break;
35         }
36     }
37 };
```

Código 1.16: Utilización de la función "handleClickEvent"

En este código se muestra la creación de un nuevo componente gráfico, «*MiComponente*», en el que se implementa el método «*handleClickEvent*» para poder gestionar los eventos relacionados con presiones sobre él. Esta función recibe como argumento una referencia constante de la clase «*ClickEvent*», donde se encapsula el tipo de subevento y parámetros relacionados con él. Dentro de esta función miembro se comprueba qué subevento concreto ha sucedido. Esto se verifica en la sentencia «*switch*», dando respuesta a los subeventos «PRESSED», «RELEASED» y «CANCEL». No es obligatorio dar respuesta a cada uno de ellos, bastará con hacerlo sobre aquel o aquellos en los que se esté interesado. Así, podemos dar respuesta al subevento «PRESSED» e ignorar el resto. En el caso concreto mostrado, se ha optado por gestionar todos, dando la posible

implementación del componente gráfico «*Button*» de TouchGFX—botón— para la transmisión de un «*click*»[7] al código de usuario. Para conseguirlo se utiliza el atributo «presionado» que inicialmente tiene el valor «false» cargado en la lista de inicialización del constructos – línea 5 –. Al producirse la presión sobre el componente se dispara el evento «*ClickEvent*» con el valor «PRESSED» como subtipo de evento, lo que hace que el atributo «presionado» pase a tener el valor «true» indicando que se ha producido la primera condición para generar un «click». Cuando se vuelve a producir el evento «*ClickEvent*», al liberar la presión sobre el componente y haberlo presionado anteriormente —presionado=true—, se transfiere la señal de ocurrencia del evento al código que el usuario provee, fuera del componente, para dar respuesta al «*click*» —presión y liberación sobre el componente—. Esto se hace mediante el mecanismo de retrollamada, no mostrado en el código anterior y que es tratado en la sección 1.11.2. En cualquier caso, el atributo «presionado» pasa a tener el valor «false» quedando en condición para detectar otro «*click*». Si por el contrario, el segundo evento «*ClickEvent*» tiene como subtipo «CANCEL» —se ha liberado la presión fuera del componente, previo arrastre fuera del área del mismo—, el atributo «presionado» pasa a tener el valor «false» cancelando la detección previa del «*click*».

No se muestra en el código anterior, pero en cualquiera de los casos de la sentencia de control «*switch*», se puede recuperar las coordenadas, relativas al componente, mediante la llamada a los métodos *getX()* y *getY()* de la referencia «*evt*» de la clase «*ClickEvent*».

La gestión del evento «*DragEvent*» se realiza de una forma similar al de «*ClickEvent*». Para poder dar servicio a este evento se han de dar los mismos condicionantes que «*ClickEvent*» y se ha de implementar el método «*HandleDragEvent*» como función miembro del componente gráfico. Ésta recibe como argumento una referencia constante de tipo «*DragEvent*», donde se encapsula la información de dicho evento. Este evento se produce desde que se presiona sobre el componente hasta que se libera la presión, realizando arrastre entre presión y liberación por la pantalla. La capa *HAL*, entre presión y liberación, produce de forma sucesiva eventos de tipo «*DargEvent*» a razón de uno por cada *tick*[8] . La información relevante que este evento da, son las coordenadas antes de arrastre en cada *tick* —*oldX* y *oldY*—, accedidas mediante los métodos *getOldX()* y *getOldY()*, y las nuevas coordenadas en cada *tick* —*newX* y *newY*—, accedidas mediante los métodos *getNewX()* y *getNewY()*. Este evento solo tiene el subtipo «DRAGGED» que no determina el mismo, por lo que no es necesario consultarlo.

Finalmente está el evento «*GestureEvent*». Es muy similar a «*DragEvent*», pero trata de dar respuesta al «gesto» de arrastrar sobre el componente con una liberación inmediata. La velocidad de este «gesto» hará que la respuesta al evento sea mayor o menor. Generalmente esta respuesta consistirá en un movimiento decelerado del componente que será proporcional a la velocidad de arrastre. Los datos que facilita este evento son las coordenadas donde sucede, accedidas mediante las funciones *getX()* y *getY()*, la velocidad del gesto, accedida mediante la función *getVelocity()*, y el tipo de evento, que puede ser «SWIPE_HORIZONTAL», para el gesto horizontal o «SWIPE_VERTICAL» para el gesto vertical, ambos accedidos mediante el método *getType()*.

[7] Se produce cuando se presiona y a continuación se libera sobre el componente.

[8] Lapso de tiempo que utiliza TouchGFX que está originado en la señal de sincronismo vertical de la pantalla.

1.11.2 Clase Callback y manipuladores de eventos de usuario

TouchGFX implementa un mecanismo de manipulación de eventos de usuario gestionados por la clase «*Callback*». El manipulador de eventos es toda función cuyo código esté destinado a dar respuesta a algún suceso —evento— acaecido en un control gráfico, como pueda ser una pulsación, una pulsación con liberación— «*click*»— o arrastre sobre la pantalla.

La clase «*Callback*» encapsula un puntero a función miembro de un objeto de otra clase distinta de *Callback*. Esto posibilita que un objeto de una clase pueda llamar a una función miembro de un objeto de otra clase. Lo más habitual es que un objeto control gráfico – *Widget*– pueda llamar a una función miembro de la vista que lo contiene y que constituya su manipulador de evento. Así, cuando un control botón es presionado, puede llamar, a través de un objeto de la clase *Callback*, a una función miembro del objeto vista –*View*– donde está contenido y que el código de dicha función responda al evento. El siguiente esquema muestra este mecanismo:

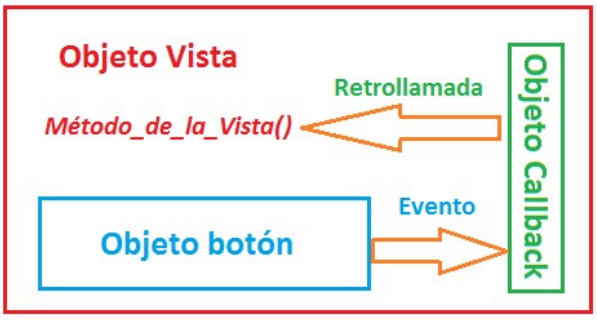


Ilustración 1.22: esquema del mecanismo de retrollamada.

Como se aprecia en este esquema, el objeto «*botón*» está contenido dentro del objeto «*Vista*», y este último posee una función —*Metodo_de_la_Vista()*— que actuará como manipulador de evento del objeto *botón* para el tipo de evento *click*. El objeto *Callback* hace de intermediario entre el botón y el método de la vista para que el primero pueda ejecutar este método de forma indirecta. El objeto *Callback* aparece contenido dentro del objeto Vista, pero el objeto botón no accede a él directamente, sino que lo hace mediante un puntero que posee internamente y que debe apuntar al objeto *Callback*. Concretamente el puntero que el objeto gráfico —botón— posee es de tipo «*GenericCallback*», clase antecesora de *Callback*. Esto es así para poder utilizar el mecanismo de polimorfismo[9].

Existen tres versiones distintas de la clase *Callback*, diferenciándose cada una de ellas en los argumentos que pasa a la función que llama —la función que actúa como manipulador de evento—. Cada objeto gráfico, dependiendo del tipo de evento que sea capaz de disparar, utilizará una de estas tres versiones, siendo responsabilidad del programador conocer cuál es la versión de la clase *Callback* que el objeto gráfico espera para gestionar dicho evento.

[9] Mecanismo que utiliza C++ y otros lenguajes orientados a objetos para que desde un puntero de la clase antecesora se pueda llamar a una función miembro de la clase derivada aún sin tener definición de la misma en la clase antecesora, aunque sí declaración – es lo que se conoce como funciones virtuales – .

Antes de continuar la explicación del mecanismo de retrollamada, conviene recordar la forma de llamada a funciones mediante punteros en el lenguaje C y la forma de llamadas a funciones miembro de un objeto mediante punteros en C++.

El siguiente código muestra la llamada a una función mediante puntero en C:

```
1  #include <stdio.h>
2  void muestra_saludo(void)
3
4  main()
5  {
6    void (*puntero_a_funcion)(void);    //declaración del puntero a función(1)
7    puntero_a_funcion = muestra_saludo; //inicializa el valor del puntero(2)
8    puntero_a_funcion();                //se desreferencia el puntero(3)
9  }                                     //también es aceptable la sintaxis
10                                       //"(*puntero_a_funcion)()" aunque se considera
11                                       //obsoleta
12
13 void muestra_saludo(void)
14 {
15   printf("Función Saludo");
16 }
```

Código 1.17: Puntero a función en lenguaje C.

Como se muestra en el código, el proceso de llamar a una función mediante un puntero necesita tres pasos; la declaración del puntero a función, la inicialización del puntero con la dirección de la función a llamar y por último, llamar a la función a través del puntero. En este ejemplo la función llamada no devuelve ningún valor y no admite ningún argumento. El puntero utilizado para llamar a esta función debe estar acorde con este prototipo. Si la función devolviera algún valor de un tipo concreto y/o admitiera valores de tipo concreto como argumento, el puntero debería tener esta misma firma.

El caso de puntero a función miembro de un objeto en C++, es similar pero tiene ciertas diferencias sintácticas:

```
1  #include <iostream>
2
3  class A
4  {
5   private:
6      int a;
7   public:
8    A(){a=10;}
9    void suma(int b) {
10     cout << "La suma es : "<< (a+b) << endl;
11   }
12 };
13
14 main()
15 {
16    void (A::*puntero_a_metodo) (int); //declaración del puntero a función miembro(1)
17    puntero_a_metodo =&A::suma;        //inicializa el valor del puntero(2)
18    A objeto_A;                        //se crea un objeto de la clase A, que contiene la
19                                       //función miembro a ser llamada(3)
20   (objeto_A.*puntero_a_metodo) (3);  //se desreferencia el puntero(4)
21 }
```

Código 1.18: Puntero a función miembro en lenguaje C++.

En este caso el proceso tiene cuatro pasos. Al igual que ocurría con el lenguaje C, en C++ se declara el puntero a función miembro de un objeto con una sintaxis similar, pero se debe especificar a la clase que pertenece esta función miembro. Esto constituye la primera gran diferencia con respecto de C. Ello se hace mediante el identificador de la clase – en este caso «A» – y el operador de resolución de alcance – «::»–, reflejado en la línea de código 16. En la siguiente línea de código se realiza la carga del puntero con la dirección

de la función miembro a ser llamada. En este paso, como en el anterior, se debe especificar el identificador de clase y el operador de resolución de alcance. Además se debe obtener la dirección de la función miembro mediante el operador «*dirección de*» – &–, algo que no era necesario en C ya que el nombre de la función ya constituye su dirección. Llegado a este punto ocurre la segunda gran diferencia del proceso respecto de C. En C++ se necesita el objeto concreto de la clase al que se quiere llamar su función miembro. Esto se realiza en la línea 18 del código, donde se crea la instancia concreta de la clase A, u objeto, llamado «*objeto_A*». Finalmente se llama a la función miembro «*suma*» del objeto «*objeto_A*» a través del puntero «*puntero_a_metodo*». Esto se hace con una sintaxis muy similar a la versión obsoleta de C —línea 20 del código—.

Con ello vemos que para poder llamar a una función miembro de un objeto se necesita conocer su clase y el propio objeto que contiene dicha función. Esto constituye un gran problema ya que los controles gráficos —*Widget*— son objetos independientes de la vista que los contiene y las funciones de manipulación de eventos serán funciones miembro de esta vista. Una posible solución sería declarar un puntero a función miembro de la clase de la vista dentro de los controles gráficos así como un puntero al objeto de la clase vista que contiene dicho objeto. Luego habría que establecer un mecanismo para pasarle al control la dirección de la función concreta de la vista encargada de la manipulación del evento y la dirección del objeto de la propia vista. Esto resolvería el problema pero limitaría la manipulación de eventos al ámbito de la vista además de tener que conocer de antemano la clase «*vista*» concreta. *TouchGFX* establece un mecanismo que independiza los controles gráficos de la vista que los contiene, de tal forma que las funciones manipuladoras de eventos pueden residir en un objeto distinto de la vista, aunque lo más habitual es que estén en ésta y no se tenga que conocer a priori la clase concreta de la vista utilizada. Los controles gráficos no tendrán conocimiento en ningún momento del objeto donde reside el manipulador del evento.

Para conseguir esto *TouchGFX* utiliza una clase que encapsula el puntero a función miembro y el puntero del objeto que contiene este miembro. Esta clase es la clase *Callback*. El esquema UML de la clase *Callback* y de su antecesora, *GenericCallback*, es el que se muestra a continuación:

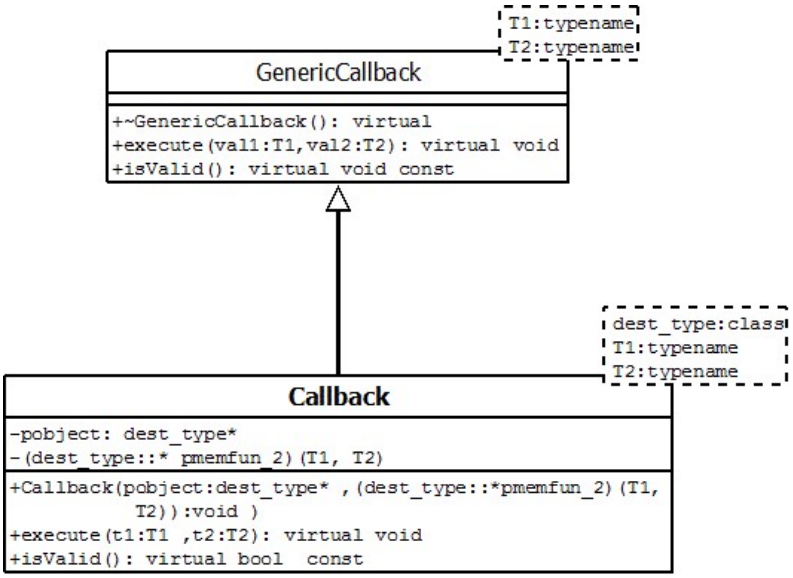


Ilustración 1.23: Esquema UML de la clase Callback.

El objeto de la clase *Callback* se declarará en la vista mientras que un puntero a la clase *GenericCallback* viene declarado en cada objeto gráfico que dispara un evento.

Como se ve en este esquema, *Callback* se trata de una clase plantilla, ya que hay ciertos tipos que maneja que no están determinados en la definición de dicha clase. Uno de estos tipos será el tipo de la clase que contiene la función miembro —*dest_type*—. Esto posibilita declarar, dentro de la clase *Callback*, un puntero al objeto que contiene la función miembro a ser llamada y el puntero a la propia función. Al tratarse *Callback* de una plantilla la declaración de estos punteros se puede realizar con un tipo indeterminado. Como toda plantilla, los tipos indeterminados quedarán finalmente determinados al crear un objeto de la clase plantilla, en este caso de la clase *Callback*. La clase *Callback* tiene tres versiones como se comentó anteriormente. La primera de ellas solo tiene un tipo indeterminado; el tipo del objeto que contiene la función a ser llamada —*dest_type*—. La segunda, dispone, además del tipo indeterminado anterior, de un tipo indeterminado adicional a ser usado como tipo de un posible argumento de la función miembro a ser llamada —*T1*—. La última versión de *Callback* posee tres tipos indeterminados; el referente al tipo del objeto que contiene la función a ser llamada —*dest_type*— y dos tipos indeterminados adicionales para ser usados como posibles tipos de argumentos a la función a ser llamada —T1 y T2—. Esta última versión de la clase *Callback* —que podemos denominar como la más completa— es la mostrada en el esquema UML anterior. La descripción de sus atributos y métodos se realiza a continuación, mostrando su código:

```
1  template <class dest_type, typename T1 = void, typename T2 = void>
2  struct Callback : public GenericCallback<T1, T2>
3  {
4      Callback() : pobject(0), pmemfun_2(0)
5      {
6      }
7
8      Callback(dest_type* pobject, void (dest_type::*pmemfun_2)(T1, T2))
9      {
10         this->pobject = pobject;
11         this->pmemfun_2 = pmemfun_2;
12     }
13
14     virtual void execute(T1 t1, T2 t2)
15     {
16         (pobject->*pmemfun_2)(t1, t2);
17     }
18
19     virtual bool isValid() const
20     {
21         return (pobject != 0) && (pmemfun_2 != 0);
22     }
23
24 private:
25     dest_type* pobject;
26     void (dest_type::* pmemfun_2)(T1, T2);
27 };
```

Código 1.19: clase Callback.

En la parte privada se observan los punteros al objeto que contiene la función miembro a ser llamada —línea 25— y el puntero a la propia función miembro —línea 26—. Los tipos de ambos punteros son tipos indeterminados especificados en la declaración de la plantilla —línea 1 (dest_type)—. En el caso del puntero a función miembro —línea 26— se dispone de dos parámetros de tipos indeterminados, especificados igualmente en la declaración de la plantilla —T1 y T2—. La clase *Callback* dispone de dos constructores, uno por defecto —línea 4—, donde los dos punteros son cargados con valores nulos, y otro donde se cargan con los valores de las direcciones a donde deben apuntar los punteros. En este segundo caso se usa el puntero interno de la clase «*this*» necesario para resolver la ambigüedad entre los nombres de los argumento pasados al constructor y nombres de los atributos internos, ya que ambos son iguales – líneas de la 8 a la 12 –. El método «IsValid» – línea 19 – permite determinar si el objeto de tipo *Callback* tiene valores válidos, es decir, que no se haya creado el objeto Callback mediante su constructor por defecto. Finalmente la clase dispone del método más importante; el método «*execute*». Este método realiza la llamada a la función miembro que constituye el manipulador de eventos. En el caso de la versión de la plantilla mostrada, se ejecuta el puntero a la función con dos argumentos

con tipos indeterminados. En el caso de la segunda versión de la clase Callback, solo se dispondrá del tipo adicional indeterminado «*T1*» como tipo a ser utilizado en el argumento en la llamada a la función miembro de la vista. Para la primera versión no existen los tipos indeterminados «*T1*» y «*T2*».

El objeto de la clase *Callback* es creado dentro del objeto que posee la función a ser llamada, en casi todos los casos, el objeto vista, es decir, el objeto de la clase *Callback* se declara dentro de la vista —estará contenido dentro de la vista—. Los pasos a seguir se muestran en el siguiente código:

```
1  #include <touchgfx/Callback.hpp>
2  #include <touchgfx/widgets/Button.hpp>
3
4  class MiVista : public View
5  {
6  public:
7    MiVista() :
8      envoltura_puntero_a funcion(this, &MiVista::Manipulador)
9      //Construcción del objeto Callback –primer enlace entre objeto "Callback"
10     //y función manipuladora de evento- paso(2)
11   {
12     miBoton.setAction(envoltura__puntero_a funcion);
13     //Se registra el objeto Callback dentro del control gráfico, -segundo enlace entre
14     //control gráfico y objeto "Callback"-  paso(3)
15   }
16   void Manipulador()
17    //Función miembro de la vista a ser llamada por el control gráfico miBoton
18   {
19     ...
20   }
21 private:
22   Button miBoton;
23   Callback<MiVista> envoltura_puntero_a funcion; //Declaración del objeto Callback, paso(1)
24 };
```

Código 1.20: mecanismo de llamada al manipulador de eventos.

En este código, que corresponde al archivo de cabecera —hpp— de la vista concreta «*MiVista*», se muestra el caso de la primera versión de la clase *Callback* —la plantilla solo tiene el tipo indeterminado del objeto que contiene la función miembro, sin argumentos para la función llamada—. Por comodidad se muestra todo el código en el archivo de cabecera, pero lo habitual es localizar las definiciones de objetos y el constructor en el archivo de cabecera mientras que el resto de código se localice en el archivo fuente —cpp—. En este caso, la definición de la función «*Manipulador*» debería ir en el archivo fuente —por supuesto, la declaración se mantiene en el archivo de cabecera—. Esta función miembro, *Manipulador*, constituirá el manipulador de evento del objeto gráfico «*miBoton*» para el evento «*click*» del mismo. En la zona privada de la vista se declara el objeto *Callback* —envoltura_puntero_a_funcion— concretando el único tipo indeterminado que posee la primera versión la clase *Callback*, en este caso se trata de la clase de la vista —*MiVista*, línea 23—. Para que funcione el mecanismo es necesario crear dos enlaces: uno entre el objeto *Callback* y el método de la vista que se ha de llamar —el que funcione como manipulador de evento del usuario; en este caso *Manipulador*—, y otro entre el objeto gráfico que dispara el evento y el objeto *Callback* que llama al manipulador. El primer enlace se lleva a cabo en el constructor de la clase *MiVista*, donde se construye el objeto *Callback* —envoltura_puntero_a_funcion— pasando como argumento a su constructor la dirección del objeto de la vista que lo contiene —*this*, línea 8— y la dirección de la función miembro a actuar de manipulador del evento del objeto «*miBoton*» —&MiVista::Manipulador—. De esta forma, el objeto de la clase *Callback* ya sabe qué método ejecutar cuando se llame a su miembro «*execute*». Ahora solo queda decirle al objeto gráfico que dispara el evento —*miBoton*— qué objeto *Callback* ha de utilizar. Para ello se procede a pasar el objeto *Callback* «*envoltura_puntero_a_funcion*» al control gráfico «*miBoton*» mediante la llamada al método «*setAction*» que poseen todos los controles gráficos para dar servicio al evento *click*. Este método recoge la dirección del objeto *Callback* y la almacena en un puntero de tipo *GenericCallback* dentro del control *miBoton*. Cuando el botón dispare el evento —se hace *click* sobre él—, llamará a la función *execute* del puntero a objeto *GenericCallback*

que posee internamente, y esta función llamará a su vez a la función «*Manipulador()*» de la vista. Todo ello sucede de forma trasparente al usuario. De lo único que se ha de preocupar éste es de crear el enlace entre el manipulador de evento y el objeto de tipo *Callback* —línea 8— y el enlace entre el objeto gráfico y el objeto *Callback* — línea 12—.

Del esquema *UML* mostrado anteriormente y de la sección de código que a continuación se muestra de *GenericCallback*, se aprecia que dicha clase antecesora de *Callback* es abstracta – posee métodos virtuales puros:

```
1  template <class T1 = void, class T2 = void>
2  class GenericCallback
3  {
4  public:
5      virtual ~GenericCallback()
6      {
7      }
8      virtual void execute(T1 val1, T2 val2) = 0;
9      virtual bool isValid() const = 0;
10 };
```

Código 1.21: clase GenericCallback.

Cuando el control gráfico *miBoton* es presionado, se desreferencia el puntero de tipo *GenericCallback* que está cargado con la dirección del objeto *Callback* de la vista y ejecuta primero el método «*isValid*», para determinar si se llama o no a la función de la vista que actúa como manipulador de evento y luego se ejecuta el método *execute* para llamar a este manipulador de evento. A pesar de que el puntero que posee el control gráfico es de tipo *GenericCallback* y dicha clase no tiene implementación de ninguno de sus métodos, al estar cargado con la dirección de una clase descendiente —*Callback*—, se llama a los métodos de esta última que tienen igual nombre que los de la clase antecesora. Este es el mecanismo implementado por C++ conocido como polimorfismo. Esto posibilita que el control gráfico no posea ninguna información de punteros, tanto del objeto vista como de la función miembro de la misma que actúa como su manipulador de evento, a excepción de un puntero a la clase *GenericCallback*. La ventaja de este mecanismo es que la declaración interna del puntero *GenericCallback* es siempre la misma con independencia de la vista donde se realice y la función manipuladora que se utilice. Internamente cada objeto gráfico declarará el puntero/s de la clase *GenericCallback* en una de sus tres versiones —con dos argumentos, T1 y T2; uno, T1; o ninguno—, dependiendo si para la manipulación del evento se requiere mayor o menor información. Cada objeto gráfico ha de facilitar una función para poder cargar la dirección del objeto *Callback* utilizado, dentro del puntero interno de tipo *GenericCallback* para el evento. En el caso se la clase *MiBoton*, esta función es *SetAction*.

Un mismo manipulador de evento puede dar servicio a eventos de distintos controles gráficos mientras que estos eventos sean del mismo tipo. Por ejemplo, se pueden poseer dos botones y establecer un único objeto *Callback* y un único manipulador para dar servicio a los *click* de ambos botones. Para conseguir esto dentro del manipulador de evento, se ha de discriminar si el evento es procedente de uno u otro botón. En este caso es necesario pasarle al manipulador de evento un argumento que identifique al botón que generó el evento. Por ello se hace obligatorio el uso de al menos la segunda versión de la clase *Callback* —aquella que pasa al manipulador de evento un argumento—:

```
1  #include <touchgfx/Callback.hpp>
2  #include <touchgfx/widgets/Button.hpp>
3
4  class MiVista : public View
5  {
6
7  public:
8    MiVista() :
9      BotonCallback (this, &MiVista::Manipulador)
10   {
11     Boton1.setAction(BotonCallback);
12     Boton2.setAction(BotonCallback);
13   }
14
15   void Manipulador(const AbstractButton &button)
16   {
17     if(&button == &boton1)
18     {
19       ...
20     }
21     if(&button == &boton2)
22     {
23       ...
24     }
25   }
26
27 private:
28   Button boton1;
29   Button boton2;
30   Callback<MiVista, const AbstractButton &> BotonCallback;
31 };
```

Código 1.22: manipulador de eventos para varios widget.

Al igual que en el caso anterior, por comodidad, se ha puesto todo el código en el archivo de cabecera de la vista —*MiVista*—. En la líneas 28 y 29 se declaran los dos botones —*boton1* y *boton2*— mientras que en la 30 está la declaración del objeto *Callback* con un tipo indeterminado; *AbstactButton*, clase antecesora de *Button*, que es la clase para la creación de *boton1* y *boton2* —el primer parámetro de la plantilla es la clase del objeto que la contiene, en este caso *MiVista*—. En la línea 9, la lista de inicialización del constructor de la vista, se procede a realizar el enlace entre el objeto *Callback* —*BotonCallback*— y el manipulador de evento que dará servicio a los dos botones —*Manipulador*—. Luego, en las líneas 11 y 12, se realiza el enlace entre cada uno de los botones y el objeto *Callback*. Finalmente, en el manipulador se procede a discriminar de qué botón se trata, comparando las direcciones de cada botón con la dirección del argumento del manipular que es pasado por el objeto *Callback* por referencia y que contiene la dirección del objeto de tipo *Button* que generó el evento. Esto es posible ya que el puntero, que internamente tiene la clase *Button* para albergar la dirección del objeto *Callback* utilizado, espera la versión con un argumento de esta clase, y cuando se genera el evento *click* dentro del botón, este pasa al objeto *Callback* su dirección como argumento —realmente es pasado como referencia—.

1.11.3 Temporizaciones

Las temporizaciones, dentro del entrono gráfico, no están pensadas para realizar tareas periódicas como sucede en el caso del sistema operativo en tiempo real, sino que están destinadas a controlar animaciones, permitiendo que un control gráfico se desplace de un lado a otro de la pantalla o aparezca, desaparezca o permanezca visible durante un tiempo. La temporización se puede realizar desde los propios controles gráficos o sobre las vistas que los contienen. El evento de temporización sucede en cada «*Tick*» del entorno gráfico, y este se dispara en la actualización de la pantalla y el cambio de búferes gráficos. Dependiendo desde donde se realice la temporización, su modo de uso será ligeramente diferente. Para la temporización desde un control gráfico, la clase antecesora —*Drawable*— de cualquiera de ellos, dispone de la función de manipulación del evento temporización llamada *handleTickEven*t:

```
1  class Drawable
2  {
3   public:
4   . . .
5   virtual void handleTickEvent() { }
6   . . .
7  }
```

Código 1.23: Declaración del evento de temporización en la clase Drawable

Este manipulador, en la clase *Drawable*, es virtual y con cuerpo vacio, lo que significa que puede sobrescribirse en las clases derivadas y en la clase base no tiene utilidad. De hecho, para el caso de temporizaciones desde controles —este caso—, este manipulador no será llamado nunca a no ser que se registre el control para poder recibir el evento de temporización. La causa de esta restricción es debida a preservar baja la carga del sistema gráfico, no ejecutando los eventos de temporización de cada uno de los controles de la vista si estos no lo necesitan. Este registro se realiza en la clase *Application*, clase que controla el par vista-presentador en uso, mediante la llamada a su método estático *registerTimerWidget*, pasándole como argumento la dirección del control a registrar:

```
1  #include <touchgfx/widgets/Button.hpp>
2  . . .
3  Button MiBoton;
4  . . .
5  Application::getInstance()->registerTimerWidget(MiBoton);
6  . . .
```

Código 1.24: Registro de la temporización de un control

En esta porción de código, se declara un control de tipo *Button*, *MiBoton*, —línea 3—, y se registra en la clase *Application* mediante la llamada a su estático *registerTimerWidget*, pasándole el objeto a registrar para que pueda recibir el evento de temporización. La llamada al método *registerTimerWidget* se hace a través del método estático *getInstance* que devuelve el objeto de tipo Application, ya que se trata de una clase de tipo *singleton*[10] [22]. Para el caso de temporizaciones de controles, *TouchGFX* permite tener hasta 16 temporizadores al mismo tiempo. Cuando ya no se necesite que un control sigua manejando el evento de temporización, es necesario eliminarlo del registro mediante el método *unregisterTimerWidget* de la clase *Application*:

```
1  #include <touchgfx/widgets/Button.hpp>
2  . . .
3  Application::getInstance()->unregisterTimerWidget(MiBoton);
4  . . .
```

Código 1.25: Eliminación del evento de temporización de un control

En los casos presentados, tanto para el registro, como para la eliminación del registro del evento de temporización, el proceso se ha realizado desde fuera del control. Lo habitual es que el propio control se registre cuando necesite gestionar el evento de temporización y se elimine del registro cuando ya no lo necesite. Para manejar el evento *Tick*, los controles disponen del método virtual *handleTickEvent*. En la siguiente porción de código se presenta el caso de un control, que deriva de la clase *Button* y se registra a sí mismo para gestionar la temporización:

[10] Patrón de diseño de clases que asegura la creación de un único objeto de la clase

```
1  class MiBoton : public Button
2  {
3    public:
4
5      MiBoton() : contador(0){ }
6
7      virtual void handleClickEvent(const ClickEvent &event)
8       {
9         Button::handleClickEvent(event);
10        Application::getInstance()->registerTimerWidget(this);
11      }
12
13     virtual void handleTickEvent()
14      {
15        if (contador < 254)
16         {
17          contador++;
18           setAlpha(255-contador);
19         }
20        else
21         {
22           visible=false;
23          touchable=false;
24          Application::getInstance()->unregisterTimerWidget(this);
25         }
26       }
27
28   private:
29
30     uint8_t contador;
31 }
```

Código 1.26: Ejemplo de clase con gestión del evento de temporización

Para el registro se necesita un evento que lo realice, generalmente se trata del evento *Click* —líneas de la 7 a la 11—. Cuando se produce un *Click* sobre el control, este procede a registrarse —línea 10— pasando a la función de registro, la propia dirección del control —puntero *this*—. Esto hace que esté en disposición de gestionar los siguientes eventos de temporización —líneas entre la 13 y la 26—. A partir de ese momento, en cada *Tick* del entorno gráfico, *TouchGFX* llamará a su evento de temporización y este irá haciendo el botón cada vez más transparente hasta su total desaparición, momento en el que el propio control se eliminará del registro de temporización —línea 24—. Este es un ejemplo poco práctico, ya que una vez presionado el botón, este desaparece para siempre —a no ser que otro control le devuelva a la visibilidad—, pero muestra de forma simple el uso del registro y eliminación del registro de temporización.

En el caso de temporizaciones gestionadas desde la vista, no es necesario registrarla para que gestione el evento *Tick*, ya que al haber una sola activa a la vez, *TouchGFX* determina que la llamada a su evento de temporización no sobrecarga demasiado el entorno gráfico. El método que manipula este evento es el mismo que para los controles: *handleTickEvent*. La manipulación de temporizaciones en la vista puede tomarse como un lugar donde se centralicen todas las animaciones, pero esto hace el código más complicado de entender, mientras que la manipulación desde cada uno de los controles, debido al encapsulamiento propio de C++ —y de los lenguajes orientados a objetos—, oculta esta complejidad. Una vez desarrollado el control, el usuario no tiene que conocer cómo funciona por dentro, solo debe conocer su comportamiento. Un uso más racional de las temporizaciones gestionadas desde la vista son aquellas destinadas a inicializaciones que deban producirse con algún tipo de retardo después de la carga de la vista. Este es el uso que se le ha dado a la única vista de este proyecto, llamada *FftView*. La manipulación que se hace sobre el evento *Tick*, en esta vista es la siguiente:

```
1  class FftView : public View<FftPresenter>
2  {
3   public:
4
5    FftView():Contadortick(1), Iniciado(false), . . .
6
7    . . .
8
9    void  FftView::handleTickEvent()
10   {
11    if(!Iniciado)
12     {
13      Contadortick++;
14      if(Contadortick%5==0)
15          {
16           presenter->obtenerModelo()->ponerAutoAjuste();
17           Iniciado=true;
18          }
19     }
20   . . .
21  private:
22
23    int32_t Contadortick;
24    bool Iniciado
25    . . .
26 }
```

Código 1.27: Ejemplo de temporización en la vista

La pretensión de la gestión de temporización en esta vista es realizar el autoajuste vertical de la gráfica una vez se haya cargado la vista al iniciar el sistema. La vista dispone de un método para inicializar los controles gráficos en ella contenidos, llamado *setupScreen* y dispone de su constructor, donde se puede llamar a los constructores de los controles, pero la manipulación de esta temporización —líneas entre la 9 y la 19— hace una inicialización que está fuera de la vista: solicita al modelo que la gráfica sea autoajustada —línea 16—. El modelo va a cambiar sus atributos para reflejar esta solicitud, pero el autoajuste se produce fuera del contexto de ejecución del entorno gráfico. El modelo traslada esta solicitud a la aplicación, que es ejecuta en otra tarea del sistema operativo, mediante el envío de un mensaje a través de colas del sistema operativo. Con todo ello, sin entrar, por ahora, en más detalle, es poner de manifiesto que se necesita una acción con un cierto retardo después que todo la vista se haya inicializado —especialmente el objeto que representa la gráfica sobre el sistema—. Para realizar esta temporización, que solicite el autoajuste, se utilizan los atributos de la vista *Contadortick* e *Iniciado*. El primero contendrá el número de *Ticks* desde el inicio de ejecución, mientras que el segundo indicará que la temporización requerida ya ha sido realizada y no se debe gestionar más *Ticks*. Este mecanismo es necesario ya que el evento *handleTickEvent* de la vista no se puede eliminar del registro de temporizaciones. Para ello, inicialmente el atributo *Contadortick* está a 1 e *Iniciado* a falso, indicando que la inicialización que se quiere realizar con la temporización de la vista todavía no se ha producido. La temporización está condicionada a que esta variable indique que no se ha producido la inicialización —línea 11—. La llamada al autoajuste se produce en el quito Tick tras la inicialización del entorno gráfico.

1.11.4 Eventos entre el Modelo, Vista y Presentador

Tal y como se ha mostrado en la sección 1.9, el patrón de programación MVC —o *MVP*— está compuesto de una vista, donde están contenidos todos los controles gráficos —*widgets*—, un modelo, encargado de mantener el estado de la información que manejan los controles gráficos de la vista y un presentador, encargado de hacer de árbitro entre el modelo y la vista, tal y como se presenta en la siguiente figura que aparece en el manual de *TouchGFX* [1]:

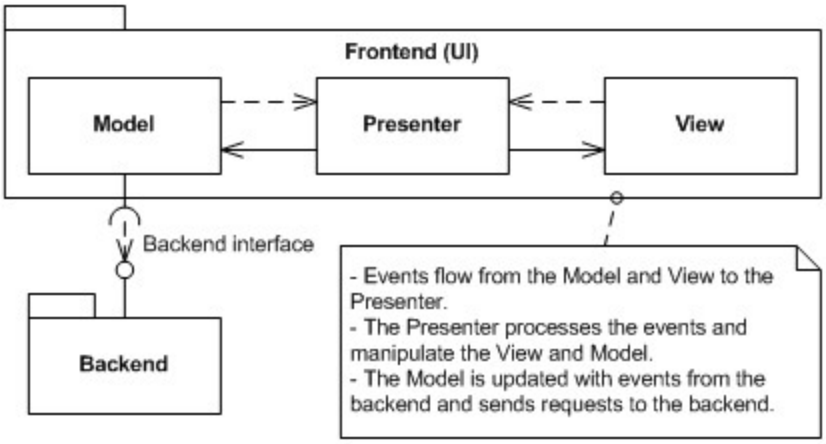


Ilustración 1.24: Esquema de comunicación del patrón MVP en TouchGFX

En esta figura se muestra el entrono gráfico constituido por el símbolo de paquete que encierra al modelo-presentador-vista, nombrado como *Fronted(UI)*, que realmente representa la clase *FrontendApplication*. Esta clase es la encargada de controlar el entorno gráfico, con atribuciones tales como la gestión del cambio de pantallas —par vista-presentador, descrito en la sección 1.12—, registrar los controles que recibirán el evento de temporización —sección 1.11.3—, y propagar los demás eventos —*ClickEvent*, *DragEvent* y *GestureEvent*—, desde la capa HAL —sección 1.11.1— hacia el control de la vista que ha de despacharlo. De hecho, desciende de la clase *MVPApplication* mostrada en el diagrama UML de la sección 1.11.1. Por otro lado está el interfaz del modelo con la aplicación en sí —la aplicación que realmente hace algo útil y que es ajena al entorno gráfico—, representada por el paquete nombrado como *Backend* en la figura. En cuanto a esta comunicación y cómo se ha de implementar la aplicación en sí, *TouchGFX* no especifica nada, con lo que este es un punto a desarrollar en este proyecto y será tratado en el capítulo 4. Aun presentando la relación entre modelo-presentador-vista que es mostrado en esta figura, se trata de un esquema muy genérico y es necesario concretar cuál es el funcionamiento de todo el ingenio. Además, el esquema idóneo para ello son los esquemas de secuencia de UML. Consultando código de ejemplo, que viene en la versión de evaluación de *TouchGFX*, se pueden deducir los cuatro posibles escenarios básicos de cooperación entre los elementos del entorno.

La primera situación se da cuando un control gráfico, o todo el entorno gráfico, no necesita tratar información que refleje el estado de este/os control/es, o mantener información al suceder los cambios de pantalla, y solo se limitan a realizar tareas gráficas —como desplegar un cuadro al ser pulsado cierto control—. Este escenario se muestra en la siguiente figura:

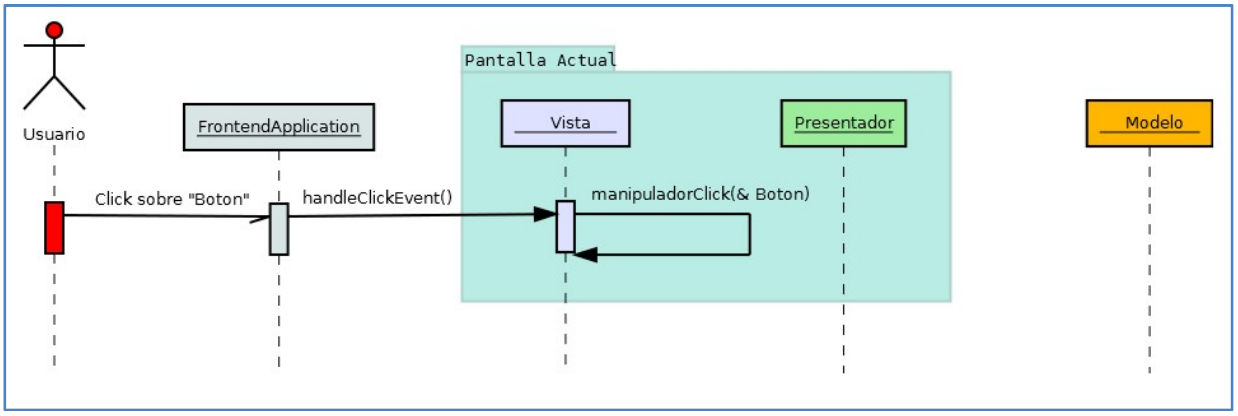


Ilustración 1.25: Tratamiento de evento exclusivamente en la Vista

En esta figura, el usuario realiza una pulsación sobre un botón, llamado *Boton*, localizado en la vista. Aunque el botón se encuentre contenido en la vista, los eventos son recogidos por el objeto de la clase *FrontendApplication* y enviados por este al control correspondiente. En este caso, se genera el evento *ClickEvent* y el objeto *FrontendApplication* lo manda al control *Boton* llamando a su método *handleClickEvent*, dedicado a gestionar este tipo de evento. Mediante el mecanismo de retrollamada —descrito en la sección 1.11.2—, el *Boton* llama al manipulador de usuario *manipuladorClick*, donde el desarrollador insertará el código que dé respuesta a la pulsación sobre *Boton*. Esta respuesta, para este escenario, será, generalmente una acción que modifique otro control gráfico, como pueda ser que se modifique el contenido de un texto que se muestra en pantalla.

En el segundo escenario entra en juego también el presentador:

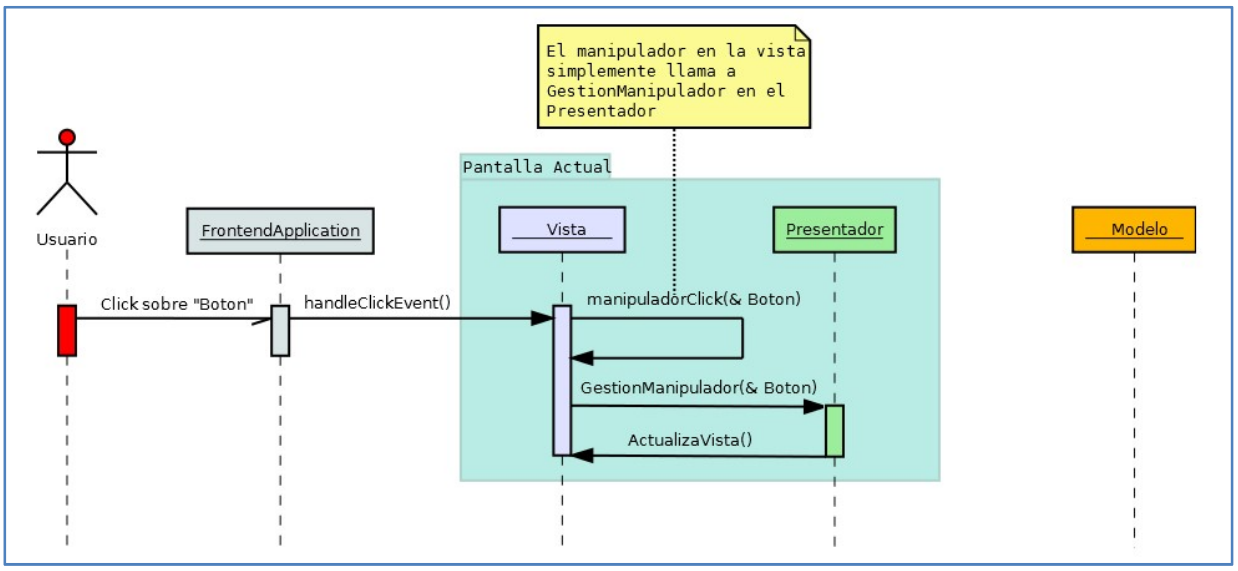


Ilustración 1.26: Delegación del evento en el Presentador

La secuencia es exactamente igual a la anterior hasta la llamada al manipulador de usuario —*manipuladorClick*—. Este manipulador se limita, exclusivamente, a llamar a una función en el

presentador —*GestionManipulador*—, que será donde se localice el código que dé respuesta al evento. Por supuesto, el argumento de este nuevo manipulador ha de ser el necesario para, al menos, identificar el control que recibe el evento. La ventaja de este nuevo sistema de manipulación, es que permite centralizar todos los manipuladores de eventos en un lugar diferente a la Vista, para que en esta solo se localice el código necesario para crear y configurar los controles gráficos. Esta centralización permite la fácil actualización de varios controles que hacen referencia al mismo estado. Así si se tiene dos controles, uno consistente en un botón con un led dibujado —que al ser pulsado conmute el estado de su iluminación—, y un botón de enclavamiento —que quede enclavado al ser pulsado, y desenclavado al ser pulsado de nuevo y así de forma sucesiva—, y ambos indican un mismo suceso —por ejemplo, el estado de un led en un GPIO—, es posible actualizar el estado del otro control al ser pulsado uno de ellos, indicando, ambos, el estado correcto. Una variante de este escenario sería aquel en el que la retrollamada no llame a ninguna función en la Vista y lo haga directamente el Presentador. Esto libera, aún más, a la *Vista* de código que no sea el puramente necesario para crear los controles. Sin embargo está técnica es raramente utilizada.

En este tercer escenario entra en juego el modelo, lo que permite mantener el estado de una pantalla —par Vista-Presentador— al realizarse la conmutación entre las distintas que componen el entorno gráfico:

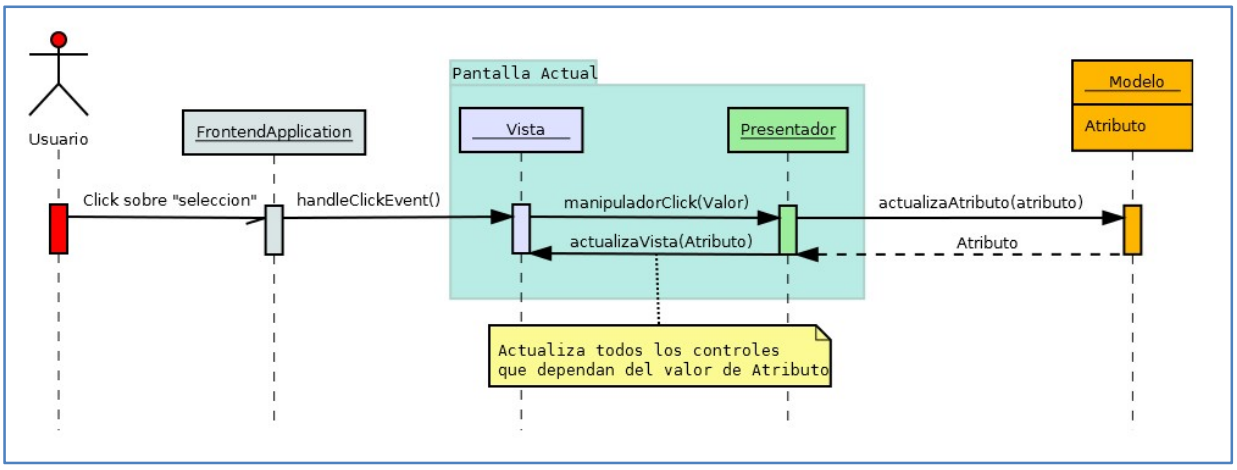


Ilustración 1.27: Actualización del Modelo desde evento en la Vista

En este caso, el usuario realiza una selección entre distintos botones de opción mutuamente excluyentes —solo puede haber uno seleccionado—, y la información de esta selección debe ser guardada en el modelo con el fin de restituir dicha selección al ocurrir el cambio entre distintas pantallas. Como en casos anteriores, al producirse la pulsación sobre el control, el evento *ClickEvent* es disparado por el objeto de tipo *FrontendApplication* y enviado al control correcto. Por simplicidad en el esquema, el manipulador de usuario se ha puesto directamente en el presentador, aunque, como se comentaba en el caso del escenario dos, lo habitual es que esté en la propia *Vista* y que desde allí se llame a algún método del *Presentador* —que es como se hace en este proyecto—. El presentador, con la información que le envía el evento de la *Vista*, actualiza un atributo en el *Modelo* —Atributo— que refleja el estado de selección —como pueda ser, reflejar el índice (cardinal) del botón seleccionado dentro del grupo que constituye la exclusión mutua—. Este atributo va a permanecer permanentemente a lo largo de la ejecución del sistema, ya que el entorno solo tiene un único modelo, y lo que cambia, al conmutar pantallas, es el par vista-presentador —marcado como paquete «Pantalla Actual» en el esquema—. Esto permite restituir los valores al volver a una pantalla de la que se había salido con anterioridad. El cambio en el valor de *Atributo* puede requerir la actualización en la *Vista* que algún otro control que refleje que esta selección se ha producido. Por ello, al solicitar el cambio de nuevo valor, por parte del *Presentador* en el

Modelo, el primero, se asegura que el cambio se ha producido satisfactoriamente mediante la obtención del retorno de este valor en la llamada al método *actualizaAtributo* del *Modelo*. Con este valor, manda actualizar los controles en la *Vista* que afecte este cambio, mediante la llamada al método *actualizaVista* de la *Vista*.

El último escenario es una matización del anterior:

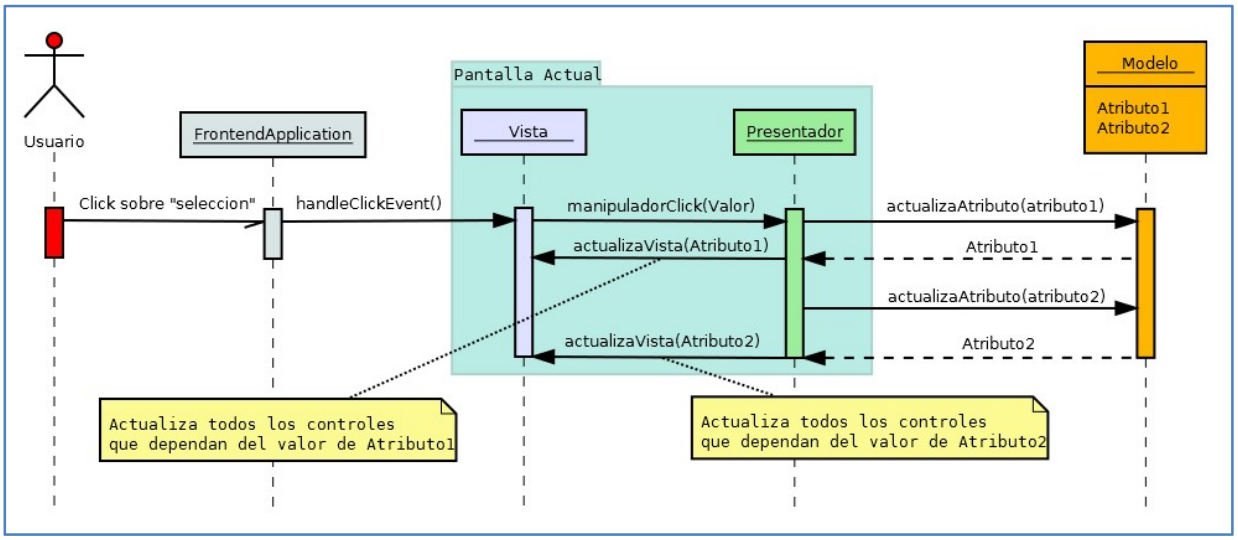


Ilustración 1.28: Varias modificaciones del Modelo por cambio en la Vista

La diferencia con el anterior, es la actualización de un segundo atributo —*Atributo2*—, debida a la actualización de uno primero —*Atributo1*—. Como en el escenario anterior, el usuario realiza una selección en un grupo de controles mutuamente excluyentes. Esto hace que el *Presentador* actualice el atributo pertinente al *Modelo* —Atributo1—. Como en el caso anterior, con la confirmación del cambio de valor en este atributo, el *Presentador* manda actualizar en la *Vista* todos los controles gráficos que dependan de este valor —*actualizaVista(Atributo1)*—. Pero existe otro atributo en el *Modelo*, Atributo2, que depende del valor del primero, con lo que el *Presentador* ha de actualizarlo acorde con el valor del atributo modificado por orden de la *Vista* —*Atributo1*—. Esto se hace mediante la llamada, por parte del *Presentador*, del método para actualizar el *Modelo*, pero con el argumento del nuevo valor para *Atributo2*. Este cambio dispara en el *Presentador* la acción de mandar actualizar en la *Vista* todos los controles que se vean afectados por este nuevo valor —*actualizaVista(Atributo2)*—. En todo este proceso, el *Presentador* ha de tener la suficiente «inteligencia» para discernir y conocer cuando el cambio de un atributo influye en otros. Esta «inteligencia» es fijada en el código por parte del desarrollador. Un ejemplo del cambio de un segundo atributo en el *Modelo*, puede darse al seleccionar un control, por parte del *Usuario*, dentro de un grupo de controles de exclusión mutua, y que esta acción necesite que se despliegue un cuadro en pantalla con cierto contenido.

En todos estos casos, los eventos tienen su origen en el mismo punto: el *Usuario*. Pero los eventos pueden ser iniciados en el modelo debido a cambios de sus atributos realizados a petición del al aplicación de usuario —la que está fuera del contexto de ejecución del entorno gráfico y que da utilidad al sistema—. En este caso, el origen de los eventos es dicha aplicación. La comunicación entre el modelo y esta aplicación es descrita en la siguiente sección.

Un problema fundamental que tiene todo este mecanismo, es el engorro de llamadas a funciones entre los distintos elementos del entorno gráfico. Por ejemplo, la Vista no conoce la existencia del Modelo, y desde ella podría realizarse la actualización de atributos directamente. El Presentador aún sería un elemento útil en la

centralización de la actualización de la *Vista* y la actualización del *Modelo*, debido a dependencias entre valores de atributos. Pero esto es un tema no especificado en la documentación de *TouchGFX* y será tema de desarrollo en el capítulo 4.

1.11.5 Eventos entre Modelo y aplicación

Estos eventos hacen referencia a la comunicación entre el entorno gráfico y la aplicación útil. El fabricante de *TouchGFX —Draupner Graphics—* no especifica nada acerca de la comunicación entre *TouchGFX* y la aplicación de usuario *—Backend—* en su manual, ni siquiera aparece código de ejemplo en los suministrados con la distribución. Afortunadamente, existe un artículo en su página web que lo trata de forma genérica.

Existen dos métodos de comunicación entre el entorno gráfico y la aplicación en sí, y ambos se realizan en el método *tick* del modelo. Este es un método que es llamado automáticamente en cada actualización de pantalla para ejecutar código de forma periódica. En el primer método de comunicación, el más simple, los periféricos hardware, desde donde se pretende obtener los datos a mostrar en el entorno gráfico o realizar el control de estos periféricos, se realiza de forma directa en el método tick. Esto tiene la ventaja de la simplicidad, pero el inconveniente de que se realiza dentro del mismo contexto del entorno gráfico, y por tanto, con una temporización de ejecución crítica. El fabricante aconseja un tiempo máximo de ejecución de un 1 milisegundo.

En el segundo método entra en juego el uso de un tarea del sistema operativo que se ejecuta de forma concurrente con la del entrono gráfico. La comunicación entre ambos se realizará de forma periódica, por tanto, se utiliza igualmente el método *tick* del modelo. La forma en cómo se realiza esta comunicación, el tipo de información que se intercambia y como se realiza la sincronización, es objeto de este proyecto y es tratado en el capítulo 4.

1.12 Conmutación entre parejas Vista-Presentador: clase FrontendApplication

Como se ha visto en la sección 1.9, el entorno gráfico está compuesto por una vista, un presentador y un modelo. Todos estos elementos —sus clases antecesoras, en el caso de la vista y el presentador— están agregados a la jerarquía de clases de la clase *Application*:

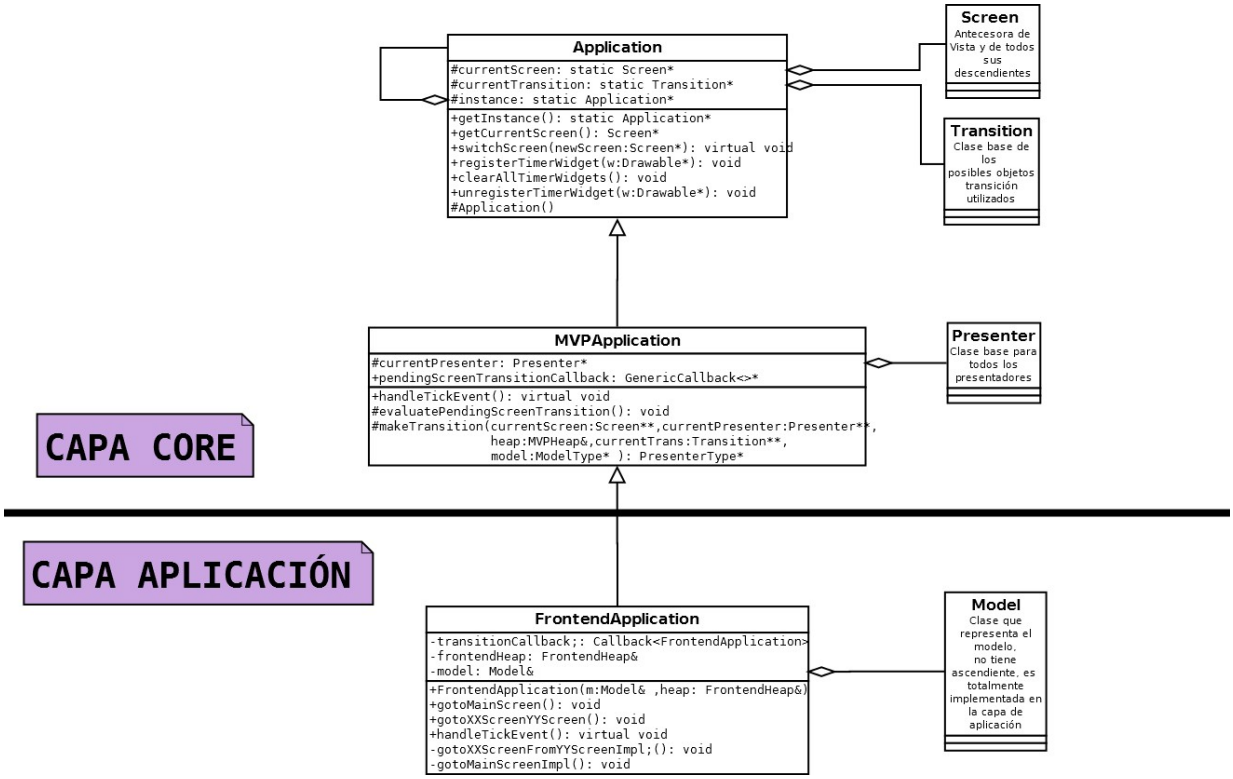


Ilustración 1.29: Jerarquía de clases de la clase Application

Los objetos de tipo vista y presentador, son subclases de *Screen* y *Presenter*, que el usuario maneja dentro de la capa de la aplicación —no mostrado en la figura—.

La clase *Application* —y sus descendientes, *MVPApplication* y *FrontedApplication*— se tratan de unas clases de tipo *singleton* [22], lo que significa que solo puede haber un objeto de la clase *Application* —o de sus descendientes—. Esto se consigue haciendo el constructor protegido, y es la llamada a la función *getInstance* la que crea el objeto al ser llamada por primera vez, y almacena la dirección del objeto recién creado en el puntero interno *Instance*. En subsecuentes llamadas a *getInstance*, solo se devuelve este puntero, sin crear un nuevo objeto de la clase *Application*.

Como se ha mostrado en la sección 1.11.1 —*Eventos desde la capa HAL hacia los componentes gráficos*— esta clase, y sus descendientes, tenía como una de sus misiones, enviar los eventos generados en la capa HAL, hacia el control gráfico que ha de recibir dicho evento en la vista actual. Otra de sus atribuciones es la de gestionar cuales de los controles gráficos han de recibir el evento *TickEvent*, que se genera en cada actualización de pantalla —sección 1.11.3—. Estas dos funcionalidades son, en cierto grado, transparentes al usuario —en el caso del evento *TickEvent*, el usuario ha de registrar los controles que han de recibirlo, sección 1.11.3—, pero existe otra funcionalidad, la consistente en el cambio entre pantallas, donde el papel del usuario es más activo.

Este cambio de pantalla consiste en la sustitución de la vista actual —que desciende de la clase *Screen*, agregado a la clase *Application*— y el presentador actual —descendiente de *Presenter*, agregado a la clase *MVPApplication*— por otro par que represente a la pantalla a la que se quiere saltar. El modelo, por el contrario, nunca es sustituido, ya que su cometido es el mantener la información de las vistas aún cuando estas no están activas, con el fin de restituir su estado cuando se vuelva a ellas —el modelo está agregado a la clase *FrontendApplication*—. Como se ve en la figura, la clase de la jerarquía que está en la capa aplicación es *FrontendApplication*, que es la hoja final en la herencia de jerarquía de la clase *Application*, y por ambas razones, es la clase con la que interactuará el usuario. Desde ella se puede acceder al resto de elementos de las clases antecesoras —*MVPApplication* y *Application*— pero teniendo en cuenta que cuando se acceda a un elemento de una de estas clases, es posible que se tenga que realizar algún tipo de conversión explícita a *FrontedApplication*. Así, por ejemplo, si se llama al método *getInstance*, para obtener el objeto *FrontendApplication*, se ha de ser consciente que lo que devuelve *getInstance* es un puntero de tipo *Application* y habrá que realizar una conversión explicita a puntero a *FrontedApplication*, de otro modo, solo se podrá acceder a los elementos de la clase *Application* y no a los de la clase *FrontedApplication*.

Para realizar el cambio de pantalla, la clase *MVPApplication* dispone de un puntero miembro, llamado *pendingScreenTransitionCallback*, de la clase *GenericCallback* sin argumentos —ver sección 1.11.2—, de una función, llamada *evaluatePendingScreenTransition*, que ejecuta el puntero *pendingScreenTransitionCallback*, si este tiene un valor válido, y el manipulador *handleTickEvent*, que llama, en cada *Tick*, a la función anterior para ejecutar el puntero:

```
1  class MVPApplication : public Application
2  {
3   public:
4      . . .
5      virtual void handleTickEvent()
6      {
7          Application::handleTickEvent();
8          evaluatePendingScreenTransition();
9      }
10
11   protected:
12     . . .
13     GenericCallback<>* pendingScreenTransitionCallback;
14
15     void evaluatePendingScreenTransition()
16     {
17         if (pendingScreenTransitionCallback && pendingScreenTransitionCallback->isValid())
18         {
19             pendingScreenTransitionCallback->execute();
20             pendingScreenTransitionCallback = 0;
21         }
22     }
23 };
```

Código 1.28: Clase MVPApplication

El puntero aparece en la línea 13 del código anterior. Este puntero es ejecutado si apunta hacia algún lugar válido y si están correctamente cargados sus punteros internos que apuntan al objeto y al método de este último que realizará realmente el cambio de pantalla —par vista-presentador—. La función que verifica si el puntero está cargado y si es así, lo ejecuta, está entre las líneas 15 a 22 del código anterior. Si finalmente lo ejecuta, a continuación, lo hace apuntar a cero para que en la siguiente llamada a la función no lo vuelva a ejecutar, ya que la carga de este puntero tiene que ser hecha por el usuario, solicitando el cambio de pantalla, y una vez realizada, no ha de volver a hacer el cambio a no ser que se vuelva a solicitar. Para verificar que el puntero ha sido cargado con una dirección válida, esta función ha de ser llamada de forma periódica, y esto se hace en el manipulador del evento *handleTickEvent* —líneas de la 5 a la 9 del código anterior—. En este manipulador, primero se llama a la versión de la clase base —línea 7—, para que la clase *Application* gestione el envío de eventos *TickEvent* a aquellos controles que han sido registro para ello, así como a la vista actual —ver sección 1.11.3—, luego, llama a la función que verifica si hay solicitado cambio de pantalla — *evaluatePendingScreenTransition*—. En este

proceso se muestra como se inicia la llamada a la función que realizará el cambio de pantalla, pero no se muestra cuál es esa función, y cómo el usuario solicita el cambio de pantalla —carga del puntero *evaluatePendingScreenTransition*—. Todo ello se realiza en la clase *FrontendApplication*. Se deben crear dos funciones, dentro de esta clase, una de ellas que cargue el puntero con la función que va a realizar el cambio de pantalla —par vista-presentador— y otra, la propia función que realmente realice este cambio. La clase *FrontendApplication* tiene una apariencia similar a la siguiente:

```
1  class FrontendApplication : public MVPApplication
2  {
3  public:
4      . . .
5      void gotoMiScreen ();
6
7      virtual void handleTickEvent()
8      {
9          model.tick();
10         MVPApplication::handleTickEvent();
11     }
12
13 private:
14     Callback<FrontendApplication> transitionCallback;
15     . . .
16     void gotoMiScreenImpl();
17 };
```

Código 1.29: Clase FrontendApplication

La función que solicita el cambio de pantalla, y que realiza la carga del puntero de la clase *MVPApplication*, es la función *gotoMiScreen* —línea 5—. La función que realmente realiza el cambio, tras ser solicitado este con la llamada a *gotoMiScreen*, es la función *gotoMiScreenImpl* —línea 16—, que será llamada por la clase *FrontendApplication* al desreferenciar el puntero, es por ello que puede ser privada. Hay que destacar el objeto *transitionCallback* de la clase *Callback*. Esta clase contiene realmente el puntero al objeto y puntero al método de este, que se quiere llamar para cambiar de pantalla, es decir el puntero a la función *gotoMiScreenImpl*. Para que el mecanismo de retrollamada funcione, la dirección de este objeto *Callback* — *transitionCallback* —, ha de ser cargada en el puntero *pendingScreenTransitionCallback* de la clase *FrontendApplication* —ver sección 1.11.2—. Todo este proceso queda clarificado en el código de estas dos funciones:

```
1  void FrontendApplication::gotoMiScreen()
2  {
3      transitionCallback = touchgfx::Callback< FrontendApplication >(this,\
4                                              &FrontendApplication::gotoMiScreenImpl);
5      pendingScreenTransitionCallback = &transitionCallback;
6  }
7
8
9  void FrontendApplication::gotoMiScreenImpl()
10 {
11     makeTransition< MiVista, MiPresentador, touchgfx::NoTransition, Model >\
12                   (&currentScreen, &currentPresenter,\
13                     frontendHeap, &currentTransition, &model);
14 }
```

Código 1.30: Funciones para cambiar de pantalla

La primera función —gotoMiScreen, en la línea 1— solicita el cambio de pantalla al par vista-presentador llamados, respectivamente, *MiVista* y *MiPresentador*. Para ello, carga los punteros internos del objeto Callback de la clase *FrontendApplication* con las direcciones del objeto *FrontendApplication* —*this*, que es el que contiene la función *gotoMiScreenImpl* que realiza en cambio—, y con la dirección de la propia función que realiza el cambio —líneas 3 y 4 del código anterior—. La dirección de este objeto *Callback* —*transitionCallback*—, que contiene los dos punteros anteriores, es asignada al puntero *pendingScreenTransitionCallback* de la clase *MVPApplication*, para que esta pueda llamar a la función de cambio de pantalla en el siguiente *Tick* —línea 5—.

Esta función ha de ser llamada por el usuario —desde el presentador actual— para solicitar el cambio de pantalla deseado.

La segunda función —*gotoMiScreenImpl*—, en la línea 9, realiza el cambio efectivo de pantalla mediante la llamada a la función de librería *makeTransition*. Se trata de una función plantilla, que además de los argumentos de entrada, requiere la determinación de tipos utilizados. Así se le ha de suministrar los tipos de vista y presentador —nueva pantalla— que se han de cargar. Estos tipos son, respectivamente *MiVista* y *MiPresentador*. También hay que suministrarle el tipo del objeto transición, una clase que controla la transición entre la pantalla que se abandona y la que se toma como actual. En el caso mostrado, esta clase es *NoTransition*, una clase que no crea ninguna animación especial al cambiar de pantalla. También es necesaria la especificación de la clase de modelo, que siempre será *Model*. Como argumentos, requiere las direcciones de los objetos que contienen la vista, el presentador, la transición y el modelo actual entre otros. Los punteros de estos objetos pertenecen a la clase *Application*, y están cargados con las direcciones de estos. Hay que recordar que la función de librería que realiza el cambio de pantalla, no pertenece a la clase *Application*, y por tanto no tiene acceso a estos punteros. Es por ello que se necesita pasárselos como argumentos.

En este proyecto no se realiza ningún cambio entre pantallas, pero este mecanismo es el utilizado para cargar la pantalla única de la aplicación.

1.13 Componentes gráficos de usuario.

Cuando se crea un proyecto nuevo en *TouchGFX*, se espera disponer de todos los componentes gráficos necesarios para poder desarrollar la aplicación. *TouchGFX* provee los componentes más habituales para un entorno gráfico como puedan ser botones, botones de verificación, botones de elección, imágenes, etiquetas de texto, contenedores, etc. Pero muy habitualmente se da la situación en la que se necesita un componente gráfico no implementado en la librería o que no tiene el comportamiento o características exigidas. Es entonces cuando se hace necesario crear nuevos componentes gráficos: los componentes gráficos de usuario.

Tal vez *TouchGFX* no tenga implementados todos los componentes gráficos equivalentes a otra librería gráfica[11], pero aunque en cada nueva versión se vayan añadiendo nuevos, siempre es necesario un mecanismo para crear componentes personalizados. En este sentido esta librería es muy versátil e implementa tres modos básicos de crear nuevos componentes: Especialización de los ya existentes, derivación desde el componente *Container* y derivación desde el componente *Widget*. Es necesario conocer el mecanismo de cada uno de estos modos ya que son utilizados en la creación de componentes para este proyecto y serán descritos en el capítulo 4 del mismo. *TouchGFX* también facilita la creación de componentes que manejen representaciones vectoriales, los componentes *Canvas Widget*, que serán tratados de forma liviana en este escrito ya que no han sido utilizados en este proyecto. Por último se dispone de unas clases especiales, denominadas *mixins* – que se podría traducir como «mezclado dentro» – y sirven para dar unas funcionalidades concretas a componentes existentes mediante el uso de plantillas.

En todos los casos está involucrado el proceso de derivación o herencia que facilita el lenguaje de programación C++. Dependiendo del componente creado, la dificultad en su desarrollo podrá ser mayor que la de la aplicación en sí. Desde este punto de vista, se puede hablar de desarrolladores o programadores de componentes y programadores de aplicación o simplemente usuarios.

[11] Se refiere al momento de inicio de este proyecto, ya que en cada subversión de TouchGFX se van añadiendo componentes nuevos.

1.13.1 Especialización de componentes gráficos existentes.

La forma más fácil y rápida de crear nuevos componentes gráficos es mediante la especialización de los ya existentes. Esto posibilita alterar el comportamiento de este componente, adaptándolo a las necesidades requeridas. Generalmente esta modificación consiste en el cambio del valor de un atributo o modificación de la gestión interna de los eventos procedentes de la capa HAL. Pero tiene el inconveniente de no ser muy versátil; solo se modifica levemente el comportamiento del componente de origen, manteniendo la esencia del mismo. Para conseguirlo se debe heredar desde el componente a modificar y cambiar, en la nueva clase heredada, la parte del código necesaria para que se comporte como se espera. Por ejemplo, si se desea crear un componente que genere eventos – y por tanto pueda responderse a ellos mediante código de usuario – cada vez que se presiona sobre él —no cuando se produce la presión seguida de la liberación—, se puede crear un nuevo componente, llamado *MiPulsador*, por ejemplo, que herede del componente botón genérico *Button*, que viene en *TouchGFX*, y modificar la gestión del evento *ClickEvent* para que genere un evento de usuario nada más ser pulsado. Una posible implementación de este componente es la mostrada en el siguiente código:

```
1  #include <touchgfx/widgets/Button.hpp>
2
3  class MiPulsador : public Button
4  {
5    public:
6      MiPulsador():Button(){}
7
8      void ponerEventoUsuario(GenericCallback< const AbstractButton& >& callback)
9       { EventoUsuario = &callback; }
10
11     void handleClickEvent(const ClickEvent& event)
12       {
13         Button::handleClickEvent(event);//se llama al evento ClickEvent
14                                         //de la clase ascendiente
15          if(event.getType()==ClickEvent::PRESSED)
16           {
17            if(EventoUsuario!=0 && EventoUsuario.isValid())
18               EventoUsuario.execute(*this);
19           }
20       }
21
22   protected:
23     GenericCallback< const AbstractButton& >* EventoUsuario;
24 }
```

Código 1.31: Especialización de un componente gráfico

En la línea 3 se produce la herencia desde el componente *Button*. La primera actuación es crear la clase heredada mediante la llamada a su constructor en la lista de inicialización del constructor de la nueva clase – línea 6 –. En la línea 11 se modifica la función *handleClickEvent*, función que es de tipo virtual en la clase *Button*, posibilitando su especialización en las clases derivadas, en este caso en la clase *MiPulsador*. Cuando se produce la modificación de un método que gestione un evento desde una clase heredada, como norma general, y en general con todo método virtual, se llamará a la versión que implementa la clase base para que se mantenga el comportamiento original, de otra forma, el único código de gestión del evento será el que se provea en la clase que hereda, siendo responsabilidad de ésta última todo el comportamiento derivado de la gestión del evento. Para el caso del código mostrado, se crea, como atributo, el puntero de tipo «retrollamada» —GenericCallback— *EventoUsuario* con un solo argumento: una referencia constate al objeto de tipo *AbstractButton* —clase base de

todos los componentes de tipo «botón»—, cuya finalidad es la de llamar al método que el usuario decida para la gestión del evento —manipulador de evento (esto se realiza en la línea 23 del código anterior)—. Para poder cargar este puntero, se debe proveer de un método público que lo haga; este se encuentra entre las líneas 8 y 9, llamado *ponerEventoUsuario*, donde el puntero es cargado con la dirección del objeto de tipo *Callback*, pasado como referencia. Se recuerda que un objeto de tipo *Callback* encapsula internamente dos punteros: uno apuntado al objeto que contiene el método a servir como manipulador de evento y otro apuntando a dicho método. La desreferencia de los mismos se produce al ejecutar el método *execute* del objeto *Callback* —o de su clase antecesora *GenericCallback*—. Esta desreferencia se lleva a cabo en la línea 18 dentro de la gestión del evento *ClickEvent* en el método *handleClickEvent*; método que la clase *MiPulsador* reimplementa para solo gestionar el subevento *PRESSED* —comportamiento como pulsador—. Para ello, primero se comprueba que el puntero de tipo *GenericCallback* apunte a una dirección válida —a un objeto de tipo *Callback*— y que los dos punteros encapsulados internamente en el objeto *GenericCallback* apunten a direcciones válidas —línea 17—–. Finalmente se llama al manipulador de evento de usuario en la línea 18.

El esquema UML sería el siguiente:

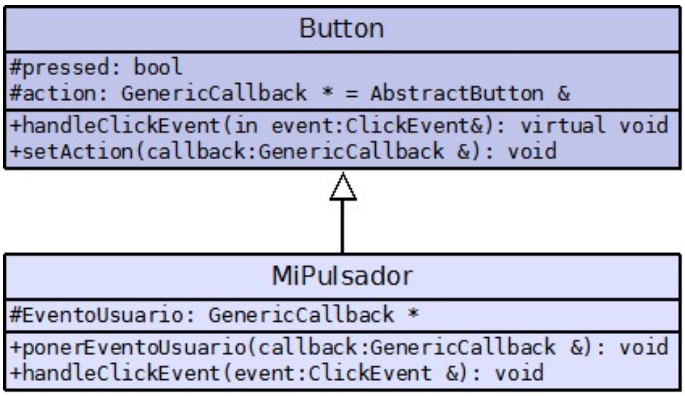


Ilustración 1.30: Esquema UML de la especialización de un componente

En la figura anterior se aprecia que la relación que tiene la clase *MiPulsador* respecto a la clase *Button* es de herencia y que es reimplementado el método *handleClickEvent* en la clase derivada —*MiPulsador*—.

Un código ejemplo de la utilización de esta clase es el siguiente:

```
1  //Código en el archivo de cabecera de la vista "MiVista.hpp"
2  class MiVista : public View<MiPresentador>
3  {
4  public:
5       MiVista():
6            CallbackUsuario(this, &MiVista::Al_Pulsador)//inicialización del objeto Callback
7            {}
8
9
10      void Al_Pulsador(const AbstractButton &button);//manipulador de evento
11
12      virtual void setupScreen();//configuración de la vista
13           /*.
14             .
15             .*/
16 private:
17      MiPulsador Pulsador;//objeto
18      Callback< const AbstractButton& > CallbackUsuario;//retrollamada
19 }
20
21 ////////////////////////////////////////////////////////////////////
22
23 //Código en el archivo fuente de la vista "MiVista.cpp"
24 #include <gui/MiVista/MiVista.hpp>
25 void MiVista::setupScreen()
26 {
27   Pulsador.ponerEventoUsuario(CallbackUsuario);
28   /*.
29     .
30     .*/
31   add(Pulsador);
32 }
33
34 void MiVista::Al_Pulsador(const AbstractButton &button)
35 {
36   //Código de usuario de manipulación del evento
37 }
```

Código 1.32: Uso de la clase especializada

Entre las líneas 1 a 19 está la definición de la vista *MiVista* mientras que entre la 25 a 37 está su declaración. En la definición se definen a su vez los atributos privados *Pulsador* —objeto de la clase *MiPulsador*, definida en el código anterior (línea 17) — y *CallbackUsuario* —objeto de la clase *Callback* para llamar al manipulador de evento (línea 18) —.

Todas las vistas —clase *view*—, y las clases derivadas de ella, disponen del método *setup* donde se configuran los objetos gráficos contenidos en ella y se añaden a la misma. Así, el objeto *Pulsador* es configurado dentro de este método en la línea de código 27 y añadido a la vista en la línea 31. Por otro lado, el objeto *CallbackUsuario* es construido en la lista de inicialización del constructor de la vista —línea 6— pasando como argumento al constructor de *CallbackUsuario* el puntero del objeto que contiene el manipulador de evento —*this*, que hace referencia al objeto de tipo *MiVista*— y la dirección de la función que actuará como manipulador de evento —*Al_Pulsador*—, que está definida en la línea 10 y declarada en la línea 34.

1.13.2 Creación de nuevos componentes derivados de la clase Container.

Otra opción, más versátil, para crear componentes nuevos, es tomar como clase de partida la clase *Container*. Esta clase se trata de un contenedor de objetos gráficos, es decir, en él se pueden ir añadiendo tantos objetos gráficos como se desee, pudiendo controlar la configuración de cada uno de ellos, siendo el responsable de «dibujar» cada uno de los objetos contenidos y como estos se muestran en el propio contenedor. El contenedor está pensado para ser utilizado en el código de usuario para agrupar ciertos controles dentro de un área de visualización en pantalla, pero si se deriva de él, se pueden crear nuevos componentes mediante composición de los ya existentes. La mayoría de los métodos que dispone el contendor para controlar los componentes añadidos son virtuales, con lo que podemos dotarles de mueva funcionalidad, como por ejemplo, modificar el modo en el que los componentes se añaden al contenedor —reimplementando el método *add*—.

Normalmente, para la gestión de eventos, se dispondrá de dos mecanismos de retrollamada, debido a que se pretenderá transmitir el evento producido en un componente contenido al exterior —al manipulador de evento de usuario—. El primer mecanismo de retrollamada consistirá en la creación de un método interno que sirva como manipulador de evento del evento a tratar que genere el componente contenido. El segundo mecanismo consistirá en la transmisión del evento al manipulador de usuario mediante un objeto retrollamada —como ya ha sido explicado—. Para ilustrar la creación de componentes a partir de un contenedor, se muestra el caso en el que se desee crear un componente que consista en un botón y según se vaya presionando, cambie el texto que aparece en él de forma cíclica entre tres posibles cadenas:

```
1  class BotonAnunciador : public Container
2  {
3      protected:
4
5          Button boton;
6          TextAreaWithOneWildcard texto;
7          Unicode::UnicodeChar cadena[15];
8          /*
9          . . .
10         */
11         char *cadenas[3];
12         Callback<BotonEstados, const AbstractButton&> RetrollamadaInterna;
13         GenericCallback< const BotonAnunciador & > * RetrollamadaExterna;
14
15     public:
16         BotonAnunciador():
17             Container(),
18              RetrollamadaInterna(this, &BotonEstados::manipuladorInterno),
19             RetrollamadaExterna(0)
20         {
21           cadenas[0]="Cadena1";
22           cadenas[1]="Cedena2";
23           cadenas[2]="Cedena3";
24           add(boton);
25           add(texto);
26          /*
27           . . .
28          */
29           //falta código para situar el botón y texto en lugar apropiado
30         }
31
32          void manipuladorInterno(const AbstractButton& source)
33         {
34             //Aquí debe ir el código que gestione el cambio de cadena
35             if (RetrollamadaExterna && RetrollamadaExterna->isValid())
36             {
37                 RetrollamadaExterna->execute(*this);
38             }
39         }
40 };
```

Código 1.33: Derivación desde la clase Container

Del código anterior cabe destacar la posesión de dos componentes, *botón* y *texto*, definidos en las líneas 5 y 6, y la agregación de estos al objeto contenedor *BotonAnunciador*, realizada en las líneas 24 y 25. También se muestran los dos objetos retrollamada utilizados; uno para la retrollamada interna que ha de ser de tipo *Callback* —*RetrollamadaInterna*, definida en la línea 12— y que sirve de unión entre el evento generado en el botón interno *botón* y el manipulador interno *ManipuladorInterno* —línea 32—, y el puntero al objeto retrollamada externa de tipo *GenericCallback* —*RetrollamadaExterna*, definida en la línea 13—, que es ejecutado en el manipulador interno cuando tiene un valor válido. Este valor válido se ha de asignar externamente al componente creado mediante un método público, tal y como se mostró para el caso de crear un nuevo componente por especialización de uno existente, de tal forma que esté cargado con la dirección de un objeto de tipo *Callback* que gestione la llamada a un manipulador externo.

La utilización de este nuevo componente creado es exactamente igual que la de cualquier otro, e igual a como se mostró en el código de utilización de un nuevo componente creado por especialización. El esquema UML de la creación del nuevo componente a partir del contenedor es:

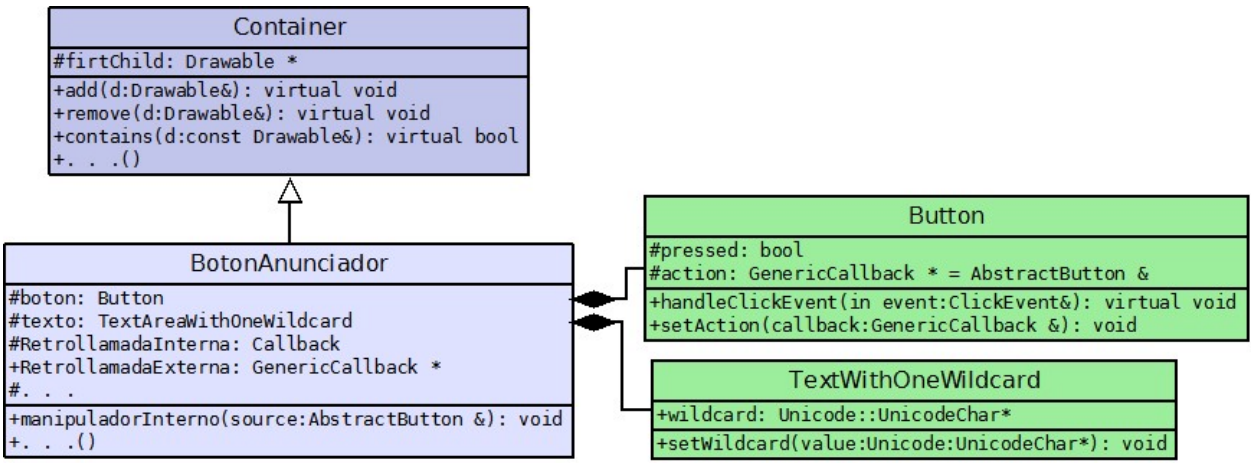


Ilustración 1.31: Esquema UML de la creación de un componente a partir de un contenedor.

En este esquema, al igual al anterior, se ilustra la relación de herencia entre la clase base —*Container*— y la derivada —*BotonAnunciador*—. Cabe destacar una nueva relación que se da en este caso: la relación de composición. Esta relación es la mantenida entre la clase *BotonAnunciador* y las clases *Button* y *TextWithOneWildcard*, en donde la instanciación de la clase *BotonAnunciador* contendrá internamente los objetos *botón* y *texto* de las clases antes mencionadas —*Button* y *TextWithOneWildcard* respectivamente—.

1.13.3 Creación de nuevos componentes derivando de la clase *Widget*.

La tercera alternativa para crear un nuevo componente es derivar directamente de la clase *Widget*, que es la clase base de todos los componentes gráficos. Esta última opción proporciona total libertad para la creación del componente, pero es la que más carga de trabajo necesita. Tal es el control que se tiene sobre el comportamiento del componente, que se debe proveer la forma en la que se dibuja en pantalla, lo que implica determinar qué se debe dibujar y donde hacerlo. Igualmente se debe conocer cuál es el área rectangular opaca

más grande del componente, es decir, la parte «solida» —no transparente ni con un índice «alpha» inferior a 255— más grande que forma parte de este. Esto último es necesario ya que, para la representación en pantalla, *TouchGFX* usa el algoritmo «*occlusion culling*»[12] para mejorar la eficiencia en la representación.

Para ello, en el nuevo componente se está obligado a implementar los métodos *draw* y *getSolidRect* ya que son métodos virtuales puros declarados en la clase *Drawable* —antecesora de *Widget*— pero no definidos. Estos métodos son utilizados para dibujar y especificar la zona sólida más grande del componente, respectivamente:

```
1  class MiComponente : public Widget
2  {
3      public:
4
5          virtual void draw(const Rect& invalidado) const
6          {
7            //implementación de las operaciones de dibujado
8          }
9          virtual Rect getSolidRect() const
10         {
11           //devolución del área sólida más grande del componente
12         }
13 };
```

Código 1.34: Estructura de un componente descendiente de Widget.

Su representación en UML es similar al caso de la especialización, solo que en este se deriva siempre de *Widget*:

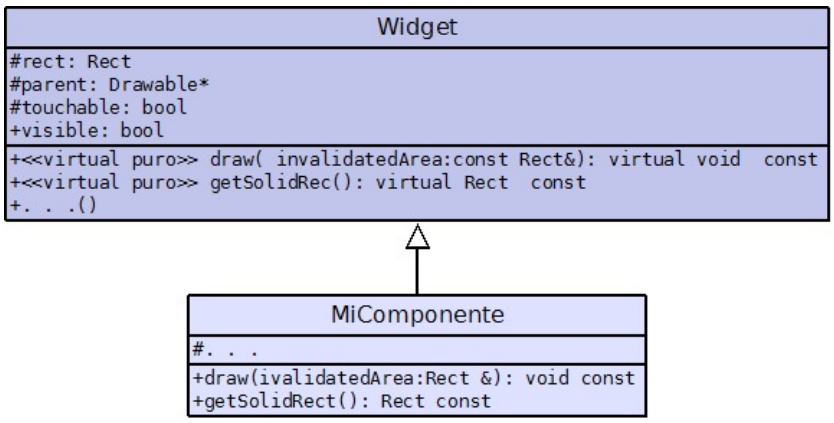


Ilustración 1.32: Esquema UML de la derivación desde el componente Widget

En este esquema cabe destacar la implementación de los métodos *draw* y *getSolidRect* en la clase derivada *MiComponente*, siendo estos métodos virtuales puros en la clase base —antecesora— *Widget*.

Antes de continuar explicando cómo se implementas estas dos funciones, es necesario conocer como se especifica un área rectangular en *TouchGFX*. Para ello, la librería dispone de la clase *Rect*, que encapsula las coordenadas de la esquina superior izquierda del área referida —«x» e «y»—. Estas coordenadas pueden estar dadas en valores absolutos, referidos a la esquina superior izquierda de la pantalla, o en relativos al componente,

[12] Podría traducirse como «ocultación selectiva» y consiste en no pintar aquellos objetos o parte de ellos que quedan ocultos por otros objetos.

en tal caso están referidas a la esquina superior izquierda del mismo. Dependiendo del contexto en donde se especifique esta área, Las coordenadas estarán expresadas en valores absolutos o relativos. Si esta área es una parte del componente, las coordenadas serán relativas. Esto también incluye el caso de especificar las coordenadas de un componente dentro de otro —como por ejemplo un contenedor—, donde las coordenadas del componente contenido estarán expresadas de forma relativa al componente contendor. En cambio, si el componente está directamente contenido en la vista, las coordenadas de dicho componente estarán expresadas de forma absoluta. Como norma general, las coordenadas de todo objeto estarán referidas al contenedor que lo contiene —ya sea un componente contenedor en sí o la propia vista—. La clase *Rect* dispone también de los atributos *width* y *height* que informan, respectivamente de ancho y alto del área. Estos atributos, así como los referentes a las coordenadas, son públicos, lo que significa que pueden ser accedidos mediante el operador de elemento a estructura. Así, si tenemos el objeto *rectángulo*, de la clase *Rect*, para acceder a su dato de altura bastará con usar *rectangulo.height*.

En ocasiones será necesario «traducir» los valores de las coordenadas relativas a absolutas. Para ello la librería dispone de las funciones *getAbsoluteRect* y *translateRectToAbsolute*, ambas como métodos de la clase *Drawable*, y por tanto disponibles para todos los componentes gráficos. La primera de ellas devuelve un objeto de tipo *Rect* con los valores absolutos de las coordenadas del componente gráfico que lo invoca; la segunda, admite como argumento de entrada-salida, una referencia de tipo *Rect*, de tal forma que después de ser llamada, el objeto *Rect* ya contenga los valores absolutos de las coordenadas del área al que hace referencia.

Cuando *TouchGFX* manda redibujar un componente, lo hace especificando las áreas del mismo en las que es necesario. Esto es así debido a dos motivos; por un lado habrá ocasiones en las que no haga falta redibujar todo el componente —por ejemplo, no será necesario redibujar aquellas partes que están ocultas por otro componente—, y por otro lado, la librería redibuja cada componente mediante subregiones del mismo. Esto posibilita dibujar varios componentes «a la vez» «multiplexando» la acción de dibujar subregiones de componentes distintos.

Por ello, cuando se crea un componente nuevo mediante la herencia desde la clase *Widget*, se ha de ser consciente, al tener la responsabilidad de su dibujado, que la librería puede solicitar redibujar parte del mismo y no su totalidad. Esta solicitud ocurre en la llamada al método *draw* de cada componente, donde la librería pasa como argumento la región a ser dibujada —objeto de tipo *Rect*—. Esta región está especificada en coordenadas relativas al componente. Sin embargo, el programador ha de redibujar esta región en coordenadas absolutas, ya que lo hará directamente sobre el buffer secundario. El programador tiene total acceso a dicho buffer, pero solo debe dibujar en la región que le ha sido requerido, en caso contrario se dibujará sobre otros componentes y el resultado puede ser impredecible. Como primera medida a ser tomada dentro de la función *draw*, será utilizar las funciones *getAbsoluteRect* o *translateRectToAbsolute*. Luego, referenciado a las coordenadas absolutas, habrá que ir barriendo el área a redibujar, poniendo en cada punto el valor correcto.

En la siguiente figura se muestra el área a redibujar —indicado por las flechas rojas— del componente de tipo botón de menú «VENTANAS» —indicado por las flechas azules—:

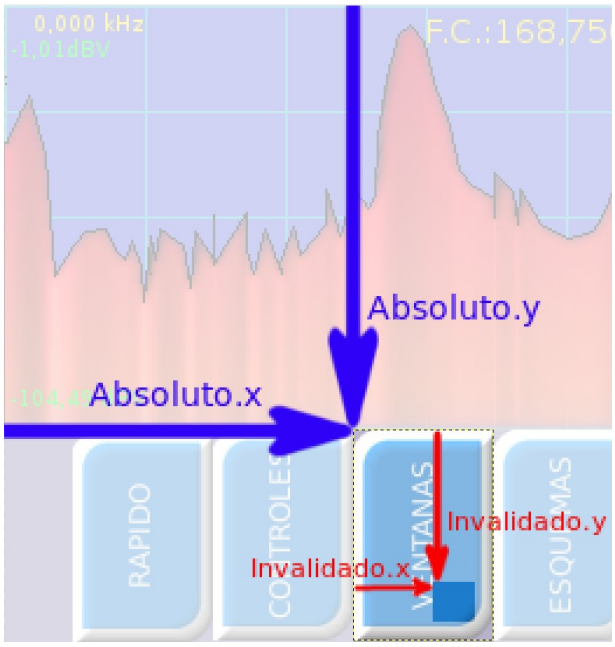


Ilustración 1.33: Esquema de utilización de la función "draw"

La librería gráfica mandará redibujar el área mostrada del componente «VENTANAS», pasándole un objeto *Rect* en coordenadas relativas, en este caso el objeto *Rect* ha sido denominado «invalidado». La posición de esta región ha de ser definida en coordenadas absolutas y para ello es necesario conocer las coordenadas absolutas del componente. Podría pensarse que las coordenadas del componente ya están expresadas de forma absoluta, pero el componente no está directamente contenido en la vista, sino que lo está en un contenedor que agrupa a todos los elementos de menú —las coordenadas del botón «VENTANAS» están referidas a la esquina superior izquierda del contenedor en el que está contenido, que en este caso, esta esquina corresponde a la esquina superior izquierda del botón «RAPIDO»—. Para determinarlas se debe hacer uso de cualquiera de las dos funciones antes mencionadas; esté será un paso que siempre se debe dar dentro de la función *draw* antes de iniciar el dibujado. Una vez obtenidas las coordenadas absolutas del componente, las coordenadas absolutas del área se determinarán sumando las coordenadas absolutas del componente más las relativas del área. El redibujado del área se realizará, normalmente, mediante un doble bucle «for» donde se recorra cada uno de sus puntos de área asignándoles los valores deseados.

Un código genérico para la función *draw* es la siguiente:

```
1  void MiComponente::draw(const Rect& Invalidado) const
2  {
3
4      Rect Absoluto = getAbsoluteRect();
5
6      uint16_t* BufferSecundario = HAL::getInstance()->lockFrameBuffer();
7
8      for (int y = Invalidado.y; y < (Invalidado.y + Invalidado.height); y++)
9      {
10       for (int x = Invalidado.x; x < (Invalidado.y + Invalidaddo.width); x++)
11        {
12         BufferSecundario[Absoluto.x + x + (Absoluto.y+y) * touchgfx::HAL::DISPLAY_WIDTH] =\
13
14                                                 Color::getColorFrom24BitRGB(0x1D,0x7C,0xCC);
15       }
16     }
17
18     touchgfx::HAL::getInstance()->unlockFrameBuffer();
19 }
```

Código 1.35: Implementación genérica del método draw.

Tal y como ya se ha comentado, la función *draw* recibe como argumento la referencia del área a ser redibujada —objeto Invalidado de la clase Rect—. Como primera acción a tomar, dentro del cuerpo de la función, se traducen las coordenadas del componente de relativas a absolutas —línea 4—. A continuación se obtiene la dirección del buffer secundario mediante la llamada a la función *lockFrameBuffer* —línea 6—. Con la llamada a esta función se consiguen dos cosas: obtener la dirección que actualmente se está utilizado como buffer secundario y asegurarse que las operaciones *DMA* —*DMA2D*— han concluido. Como ya ha sido explicado, en la representación en pantalla por parte de la tarea *taskEntry* —método de la capa *HAL*—, se intercalan las acciones del periférico *DMA2D* —moviendo las imágenes, en formato mapa de bits, o parte de ellas desde la ROM a la a pantalla—, con el redibujado de aquellos componentes en los que no es suficiente con la acción del *DMA* —como pueda ser la representación de una gráfica que cambia con el tiempo— cuya acción es asumida directamente por la ejecución del corazón del microcontrolador. Es necesario que ambos procesos no accedan a la vez al buffer secundario. El método *lockFrameBuffer*, al ser llamado, queda bloqueado hasta que el *DMA2D* ha terminado de realizar su labor, además bloquea el acceso buffer, por ello, cuando retorna el valor de la dirección del buffer, se puede estar aseguro que hay plena libertad para dibujar directamente sobre el mismo. Este método —*lockFrameBuffer*— pertenece al objeto de tipo HAL que es accedido a través su método estático *getInstance*. Esto se hace así debido a que la clase HAL sigue el patrón de programación «*singleton*»[13] [22]. Luego, se barre el área a redibujar mediante el doble bucle «for». El bucle más externo barre en la dirección vertical —desde la coordenada «y» de inicio del área hasta su final vertical—, mientras que el bucle más interno barre, para cada valor vertical del bucle externo, cada uno de los valores horizontales del área —líneas de la 8 a la 16—. Debido a que el buffer está especificado como un «array» unidimensional, el acceso a cada punto del mismo se realizará multiplicando todo el ancho de la pantalla por la coordenada vertical absoluta más la coordenada horizontal absoluta del punto a pintar. Ambas coordenadas son el resultado de sumar a las coordenadas relativas del punto a tratar, dentro del área invalidada, las coordenadas absolutas del componente al que pertenece el área —líneas 12 a 14—. En este caso, ya que el área a pintar es enteramente de un solo color, cada punto de la misma es rellenado con el valor "RGB" de este color —línea 14—. Lo último que se hace en la función *draw* es desbloquear el buffer —línea 18—.

[13] Patrón que asegura que solo exista un objeto o instancia de clase, para ello su constructor no ha de ser público.

Finalmente queda determinar el área sólida más grande del componente. Esta puede ser una labor muy ardua y puede llevar más tiempo el cálculo de la misma que el repintado de todo el componente. Una versión genérica para la función *getSolicRect* puede ser la siguiente:

```
1  Rect MiComponente::getSolidRect() const
2  {
3    if (alpha == 255)
4      return Rect(0,0,getWidth(),getHeight());
5    else
6      return Rect(0,0,0,0);
7  }
```

Código 1.36: Función getSolicRect de la clase Widget

Si el componente es totalmente sólido —valor alfa igual a 255— las dimensiones del área sólida máxima devuelta son las de todo el componente; en cambio, si existe algo de translucidez, se devuelve un área nula indicando que el componente no tiene ninguna parte sólida. En el caso de tener un componente que tenga valor alfa igual a 255 pero que haya partes del área a dibujar donde no lo haga, dejando ver parte del fondo —como en el caso de la traza de una gráfica—, en la función *getSolidRect* se puede devolver un área nula, ya que la casi totalidad del componente será transparente.

1.13.4 Componentes de tipo *Canvas* —Lienzo—.

Otra forma de crear nuevos componentes es a través de la derivación desde la clase *CanvasWidget*. Este procedimiento está recomendado para aquellos casos en los que se creen formas geométricas que deban aparecer en pantalla, es decir, que no sean el mero dibujado de un mapa de bits ya almacenado en memoria. En este sentido podemos catalogar este tipo de representaciones como «vectoriales», donde la forma geométrica es generada en una región de memoria establecida, con un sistema de coordenadas fraccionarias, para luego ser «rasterizada»[14] y pasada a pantalla —con el sistema de coordenadas de esta— .

La librería ya viene con unos cuantos componentes de este tipo como son *Shape*, que dibuja un polígono especificando sus aristas o *Line* que dibuja una línea especificando los puntos extremos, entre otros. En cualquier caso, ya sea utilizando componentes *CanvasWidget*, que pertenecen a la librería, o de nueva creación, se necesita un área de memoria donde realizar la representación. Para ello se utiliza la clase *CanvasWidgetRenderer*, una clase con todos sus métodos y atributos estáticos —no se necesita objeto o instancia—, que encapsula un puntero al «array» de memoria utilizado para la representación. Este «array» se debe declarar de forma externa a esta clase —especificando su tamaño en bytes—, para luego pasar su dirección a la clase *CanvasWidgetRenderer* mediante su método de clase *setupBuffer*. Tanto la declaración del array como su carga en la clase *CanvasWidgetRenderer,* se realiza en la función *main* del proyecto —como se sugiere en el manual de *TouchGFX*—.

[14] La imagen vectorial es convertida en mapa de bits

En el proceso de utilización o creación de un componente de tipo *CanvasWidget* están involucradas varias otras clases. Para su mejor comprensión, se presenta el siguiente esquema UML:

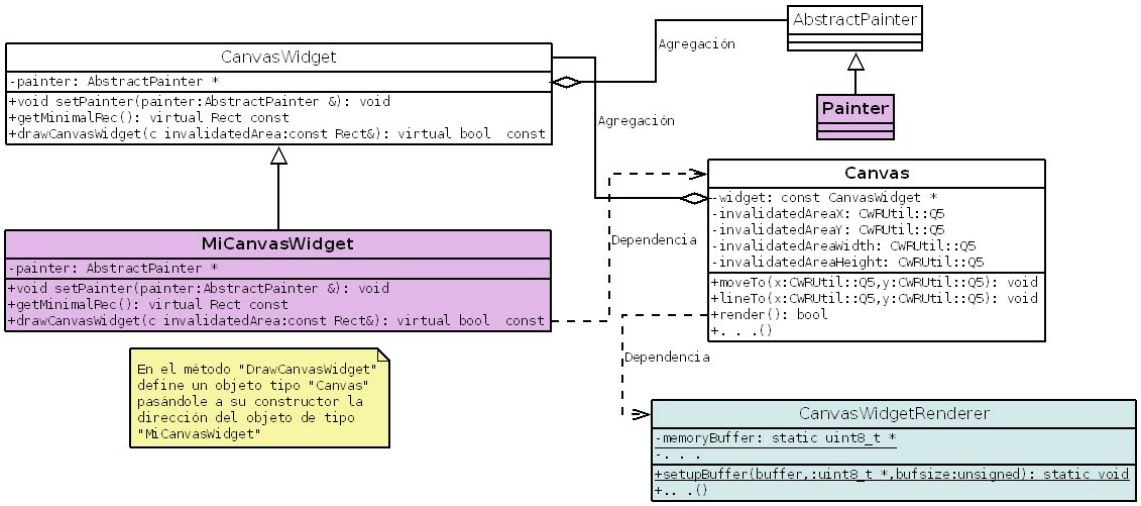


Ilustración 1.34: Esquema UML de la clase CanvasWidget y relación con otras clases

En este esquema se ve el componente principal *MiCavasWidget* —en color morado— que deriva de la clase abstracta *CanvasWidget*. Si *MiCanvasWidget* es un componente ya existente en la librería, solo se ha de preocupar de usar sus métodos ya implementados, pero si por el contrario se trata de un componente de usuario, se ha de suministrar código para los métodos virtuales *getMinimalRect* y *drawCanvasWidget*. En el primero de ellos ha de retornar el área rectangular —objeto de tipo *Rect*— más pequeña que contenga a la figura geométrica. Esta información será utilizada por el componente para realizar transformaciones de la figura —escalado, rotación o desplazamiento— de forma eficiente. Una forma sencilla de especificar esta área es hacerlo retornando el área que encierra a todo el componente —representada en la propiedad *rect*–, pero en este caso la representación no será tan eficiente. En el segundo método se ha de hacer uso de la clase *Canvas* que dispone de funciones de dibujado sobre el buffer encapsulado en *CanvasWidgetRenderer*. Por ello, la relación entre *Canvas* y *CanvasWidget* es de dependencia. El objeto de tipo *Canvas* se ha de crear dentro del método *drawCanvasWidget*, usar los métodos de dibujado del objeto *Canvas* para finalmente llamar a su método *render*. Al crear el objeto *Canvas* se pasa a su constructor la dirección del objeto *CanvasWidget* que lo crea, de este modo se establece la relación de agregación del segundo hacia el primero. Por otro lado se necesita de una entidad que gestione el color y forma de relleno de las figuras geométricas. Esto es realizado por un objeto de la clase descendiente de *AbstractPainter* —en el esquema anterior esta clase es *Painter* representada con color morado—. Esta clase puede ser creada por el usuario, para necesidades concretas, pero la librería ya trae consigo varias clases de este tipo como son *PainterBW*, *PainterRGB565* y *PainterRGB565Bitmap* que cubren la mayoría de las necesidades.

En este proyecto no se ha utilizado componentes de este tipo ni se ha creado ninguno nuevo descendiente del mismo, por lo que no se profundizará más en su explicación.

1.13.5 Plantillas Mixins.

Son clases que sirven para dar ciertas funcionalidades a clases ya existentes. Su cometido es la de «mezclar» la funcionalidad de una clase de origen con las propiedades de la clase *mixin* concreta. Para ello, las *mixins* utilizan la herencia y el uso de plantillas. Desde este punto de vista se pueden ver como un modificador de tipo. Así, por ejemplo, existe el *mixin* para dar funcionalidad de respuesta al evento *click* en aquellos componentes que no disponen de él. El código simplificado de este *mixin* es el siguiente [3]:

```
1  template<class T>
2  class ClickListener : public T
3  {
4  public:
5       ClickListener() : T(), clickAction(0)
6      {
7          T::setTouchable(true);
8      }
9      virtual ~ClickListener() { }
10
11     virtual void handleClickEvent(const ClickEvent& event)
12     {
13         T::handleClickEvent(event);
14         if (clickAction && clickAction->isValid())
15         {
16             clickAction->execute(*this, event);
17         }
18     }
19     void setClickAction(GenericCallback< const T&, const ClickEvent& >& callback)
20     { clickAction = &callback; }
21
22 protected:
23     GenericCallback<const T&, const ClickEvent&>* clickAction;
24 };
```

Código 1.37: Mixin ClickEvent

En la línea 1 se especifica que se trata de una clase plantilla con un tipo indeterminado especificado como «T». Se recuerda que una plantilla puede ser tanto de un método —función miembro— o una clase. En este último caso, se debe especificar, antes de la definición de la clase, que se trata de una plantilla y cuantos tipos indeterminados y qué nombres se usarán dentro de la misma. A partir del momento de la especificación de plantilla, se podrán usar los nombres dados de tipos indeterminados como si de tipos del lenguaje se tratase. Estos tipos serán determinados en la declaración de la clase con el modificador de la plantilla de este *mixin*. Como cosas a destacar del código anterior está la llamada al constructor del tipo indeterminado, dentro del constructor de la plantilla —línea 7— y la implementación del evento *ClickEvent* —líneas de la 11 a la 18—. Como atributo privado está el puntero retrollamada *clickAction* que es asignada con el método *setClickAction* —línea 19— y ejecutado en la gestión del evento *EventClick* —línea 16—. Cabe destacar que la gestión que se hace sobre el evento *EventClick* en este *mixin* es genérico en el sentido que no discrimina si se ha presionado o liberado sobre el componente o si se ha cancelado el evento.

Para el uso de esta clase, basta con declarar el *mixin* seguido con el especificador de plantilla, determinando el tipo/clase a dar nueva funcionalidad. Si por ejemplo se quiere dar a una imagen la posibilidad de gestionar el evento *EventClick* —que por defecto no tiene—, la declaración sería:

```
1  #include </touchgfx/mixins/ClickListener.hpp>
2  #include <touchgfx/widgets/Image.hpp>
3
4  ClickListener < Image > imagenClick;
```

Código 1.38: Declaración de un mixin

Primero se encuentra la especificación de la clase *mixin* —*ClickListener*— luego, la clase a la que dar la funcionalidad del *mixin* —*Image*—, declarada dentro del especificador de plantilla —«< >»—, y por último el objeto *imagenClick*.

1.14 Componentes gráficos

1.14.1 Componentes incluidos en la librería

Los componentes gráficos incluidos en la librería de *TouchGFX* están divididos en tres grupos: *Widgets*, *Containers* y *Mixins*. Los dos primeros ya han sido descritos de forma general en la sección 1.6. El tercer grupo, *Mixins*, se trata de unas plantillas que otorgan una funcionalidad concreta a los componentes que los utilizan —ver sección 1.13.5—. También se dispone de componentes relacionados con gráficos vectoriales —ver sección 1.13.4—, pero no serán tratados en este documento ya que no han sido utilizados para este proyecto.

1.14.1.1 Widgets

Los *Widgets* o componentes gráficos simples que están disponibles en la versión de *TouchGFX* utilizada son:

- AbstractButton
- AnimatedImage
- Box
- Button
- ButtonWithIcon
- ButtonWithLabel
- Image
- Keyboard
- RadioButton
- RadioButtonGroup
- ScalableImage
- SnapshotWidget
- TextArea
- TextAreaWithOneWildcard
- TextAreaWithTwoWildcards
- ToggleButton
- TouchArea

Hay que aclarar, antes de realizar ningún otro comentario, que el aspecto físico de estos controles no es fijo o determinado. Su aspecto está establecido por los mapas de bits que se asignan para cada estado del control —

si el control tiene varios estados y tiene representación gráfica—. Existen controles sin esta representación gráfica como son *TouchArea* y *RadioButtonGroup*. *TouchArea*, consiste en una región rectangular en pantalla que puede administrar los eventos *Click* —pulsación y liberación— y *Drag* —arrastre—. Para este proyecto, se ha creado una clase derivada de esta, llamado *TouchAreaMio* —ver sección 4.4.1.2—, que sobrescribe el manipulador de evento *handleClickEvent*, para poder gestionar la situación de cancelación de la pulsación —presión sobre algún componente contenido en esta área y sin despulsar, salir de los límites de este control, pero permanecer dentro del la región que delimita el objeto de tipo *TouchAreaMio*—, y el manipulador *hadleDragEvent*, ya que esta región estará encima de la gráfica, donde se representará la FFT, y se quiere que gestione el desplazamiento horizontal de esta, al producirse el evento «arrastre» –*Drag*—. *TouchArea* deriva, a su vez de *AbstractButton*, clase abstracta que sirve como base de varios controles que gestionan el evento *Click*, principalmente de la clase paradigmática *Button* —que representa un botón en pantalla y puede disparar un manipulador de evento de usuario, al ser presionado y liberado—. La clase *RadioButtonGroup*, es la otra clase sin representación grafica y establece los grupos que engloban a los botones de opción —*RadioButton*— mutuamente excluyentes. Por otro lado, las clases más simples, que no gestionan ningún evento, son *Box* e *Image*. *Box*, muestra un rectángulo relleno de un color con capacidad de tener cierto grado de transparencia, mientras que *Image* muestra, en un área rectangular, el *bitmap* especificado. Si no se determina tamaño para este último control, se ajustará al del *bitmap* que representa, en caso contrario lo recortará, mostrando únicamente aquella parte de la imagen que entra dentro del área especificada. Si lo que se quiere es que el *bitmap* se ajuste al área que establece el control, aumentando la imagen al aumentar el control y viceversa, la clase apropiada es *ScalableImage*, que, aunque similar a *Image*, no deriva de esta, lo hace de *Widget*. Como clase icónica está *Button*, que representa un botón, y todos sus descendientes: *ButtonWithIcon*, *ButtonWithLabel*, *RadioButton* y *ToggleButton*. Son muy similares entre sí. *ButtonWithIcon* tiene la particularidad de poder representar, dentro del botón, una imagen a modo de icono, mientras que *ButtonWithLabel* permite contener una cadena de texto dentro del mismo. *RadioButton* ya ha sido comentado, y *ToggleButton* es un botón que se enclava y desenclava, de forma alternativa, en cada pulsación. En este proyecto se han utilizado *Button*, *ButtonWithLabel*, *RadioButton*, en varios lugares y *ToggleButton* como parte constituyente de la creación de la nueva clase *ElementoMenu* —ver sección 4.4.1.9—. Los controles encargados de mostrar textos en pantalla son *TextArea*, *TextAreaWithOneWildcard* y *TextAreaWithTwoWildcards*. El primero, *TextArea*, muestra un texto fijo, extraído de la base de textos —ver sección 1.4.2—, con la tipografía especificada el ella —tipo de letra y tamaño—. *TextAreaWithOneWildcard* y *TextAreaWithTwoWildcards*, a pesar de tener que usar una entrada en la base de datos de textos, contienen una cadena de texto variable, para el caso del primer control, y dos cadenas de texto variable, para el caso del segundo control. La entrada en la base de textos sigue siendo necesaria para poder definir el tamaño y tipo de letra. Para su funcionamiento, ambos controles hacen uso de la clase *Unicode*, clase que contiene, internamente, la cadena, y permite hacer operaciones tales como copiar o concatenar cadenas. En este proyecto se ha utilizado, exclusivamente, textos variables de una sola cadena, es decir, solo se ha utilizado *TextAreaWithOneWildcard*. Por otro lado, *SnapshotWidget*, permite realizar una captura de lo mostrado en pantalla y almacenarlo, en forma de *bitmap*, en el tercer buffer de pantalla. Es utilizado, sobre todo, para realizar las posibles animaciones en el cambio de pantallas, utilizado por la clase *Transition*. Finalmente está la clase *Keyboard*, que permite tener en pantalla un teclado a modo de los mostrados en los actuales móviles —teniendo en cuenta que su apariencia en siempre configurable a través de los *bitmaps* suministrados—. Este control no ha sido utilizado en este proyecto.

Para comprender el uso de estos componentes, se van a presentar dos ejemplos. La representación de eventos a través de la clase *Callback* —ver sección 1.11.2—, va a seguir el primer esquema de secuencia de eventos en el patrón *MVP*, mostrado en la sección 1.11.4 —eventos tratados exclusivamente en la vista—.

En el primer ejemplo se va a mostrar el uso de los controles *Box*, *Button*, *TextArea*, *TextAreaWithOneWildcard* —junto con la clase *Unicode*—, y la clase *Callback*. La pretensión de este primer ejemplo es presentar un fondo de color en pantalla, y encima de él, un título y un botón que al ser presionado, cambia, de forma alternativa, un texto variable que aparece en pantalla. El código está repartido entre el archivo de cabecera de la vista —donde está la declaración de la clase de la misma—, y el archivo fuente —donde están las definiciones de sus métodos—. El archivo de cabecera es el siguiente:

```
1  #include <touchgfx/widgets/Box.hpp>
2  #include <touchgfx/widgets/Button.hpp>
3  #include <touchgfx/widgets/TextArea.hpp>
4  #include <touchgfx/widgets/TextAreaWithWildcard.hpp>
5  #include <touchgfx/Color.hpp>
6  #include <touchgfx/Unicode.hpp>
7  #include <touchgfx/Callback.hpp>
8
9  class MiVista : public View<FftPresenter>
10 {
11  public:
12    MiVista():BotonCallback(this, &MiVista::Al_Boton){}
13    void Al_Boton(const AbstractButton &button);
14    void setupScreen();
15
16  private:
17    Box Fondo;
18    Button MiBoton;
19    TextArea MiTexto;
20    TextAreaWithOneWildcard MiTextoVariable;
21    Unicode::UnicodeChar CadenaVariable[20];
22    Callback<MiVista, const AbstractButton &> BotonCallback;
23 }
```

Código 1.39: Cabecera de la vista el primer ejemplo de controles Widgets

Lo primero que tiene que aparecer en el archivo de cabecera de la vista son todas aquellas inclusiones de archivos de cabecera necesarios para poder usar cada uno de los *Widgets* deseados, así como otras clases usadas —líneas de la 1 a la 7—. Los archivos de cabecera de los cuatro *Widgets* usados aparecen entre las líneas 1 a 4, mientras que las clases de utilidad —*Color*, *Unicode* y *Callback*— aparecen entre las líneas 5 a 7. En la parte privada de la clase, se declaran los controles utilizados, que son *Fondo*, *MiBoton*, *MiTexto* y *MiTextoVariable* —de los tipos respectivos, *Box*, *Button*, *TexArea* y *TextAreaWithOneWildcard*—, y los objetos de las clases de utilidad *CadenaVariable* y *BotonCallback* —de las clases respectivas *Unicode* y *Callback*—. En la parte pública se encuentra el constructor de la vista —*MiVista()*—, donde irá todo aquel código que deba ser inicializado antes de la inicialización de los controles o aquellas llamadas a los constructores con argumentos de estos. En este caso, en la lista de inicialización del constructor de la vista se construye el objeto de tipo *Callback* —*BotonCallback*—, llamando al constructor de este último, para que apunte al manipulador de evento de usuario. En la zona pública, también aparece este método que actuará como manipulador de evento —*AlBoton*— y el método de configuración de los controles gráficos que toda vista ha de tener: el método *setupScreen*:

```
1  #include <gui/Mi_screen/MiVista.hpp>
2  #include <texts/TextKeysAndLanguages.hpp>
3  #include <BitmapDatabase.hpp>
4
5  void FftView::setupScreen()
6  {
7   Fondo.setPosition (0, 0, HAL::DISPLAY_WIDTH, HAL::DISPLAY_HEIGHT);
8   Fondo.setColor(Color::getColorFrom24BitRGB(0x00, 0x00, 0xFF));
9
10  MiBoton.setBitmaps(Bitmap(BITMAP_BOTON_ID), Bitmap(BITMAP_BOTONPULSADO_ID));
11  MiBoton.setXY(HAL::DISPLAY_WIDTH/2-MiBoton.getWidth()/2,100);
12  MiBoton.setAction(BotonCallback);
13
14  MiTexto.setTypedText(TypedText(T_TEXTO_TITULO));
15  MiTexto.resizeToCurrentText();
16  MiTexto.setXY(HAL::DISPLAY_WIDTH/2-MiTexto.getWidth()/2,16);
17  MiTexto.setColor(Color::getColorFrom24BitRGB(0x00, 0xFF, 0x00));
18
19  MiTextoVariable.setWildcard(cadenaVariable);
20  MiTextoVariable.setTypedText(TypedText(T_TEXTO_VARIABLE));
21  Unicode::strncpy(cadenaDesHor, (char*)"Primera cadena", 30);
22  MiTextoVariable.resizeToCurrentText();
23  MiTextoVariable.invalidate();
24  MiTextoVariable.setXY(HAL::DISPLAY_WIDTH/2-MiTextoVariable.getTextWidth()/2,58);
25  MiTextoVariable.setColor(Color::getColorFrom24BitRGB(0xFF, 0xFF, 0x00));
26
27  add(Fondo);
28  add(MiBoton;);
29  add(MiTexto);
30  add(MiTextoVariable);
31 }
```

Código 1.40: Función setupScreen de la vista del primer ejemplo del uso de controles Widgets

La definición de este método se encuentra en el archivo fuente de la vista. En él se configura el tamaño y color del control de tipo *Box* que actuará de fondo —líneas 7 y 8—, la apariencia, posición y manipulador de evento del control de tipo *Button* —líneas de la 10 a la 12—, la cadena de texto, posición y color del texto fijo de tipo *TextArea* que actuará de título —líneas de la 14 a la 17—, y posición, color y objeto de tipo Unicode que usará el texto variable de tipo *TextAreaWithOneWildcar* —líneas de la 19 a la 25—. Finalmente, en el método *setupScreen* de la vista, se han de añadir cada uno de estos objetos a la misma mediante el método *add*, en el orden de solapamiento —orden «z»— en el que se quiere que aparezcan. Así el objeto *Fondo*, al ser añadido el primero, será el que se encuentre más abajo en el orden de solapamiento, mientras que *MiTextoVariable*, será el que aparezca más arriba, ya que es el último en añadirse a la vista. Hay que destacar el uso de clases de utilidad, como son Color —líneas 8, 17 y 25—, clase con sus métodos estáticos, que a través de su método *getColorFrom24BitRGB*, se obtiene el color en formato RGB565 a partir del color especificado en RGB888, y *Unicode* —línea 21—, clase que representa una cadena con caracteres Unicode y que dispone de métodos de manipulación de cadenas como es *strcpy* que copia una cadena en el propio objeto de tipo Unicode. Los archivos de cabecera incluidos son: el de la propia vista, donde está la declaración de la misma, la base de cadenas de texto utilizadas —línea 2—, y la base de imágenes —líneas 3—.

Otro método que ha de ir en el archivo fuente de la vista es el manipulador de evento de usuario:

```
1  void MiVista::Al_Boton(const AbstractButton &button)
2  {
3    static pasada =0;
4    if(&button==&MiBoton)
5    {
6           if(pasada%2)
7           {
8            Unicode::strncpy(cadenaDesHor, (char*)"Segunda cadena", 30);
9           }
10          else
11          {
12           Unicode::strncpy(cadenaDesHor, (char*)"Primera cadena", 30);
13          }
14     MiTextoVariable.resizeToCurrentText();
15     MiTextoVariable.invalidate();
16     pasada++;
17   }
18 }
```

Código 1.41: Manipulador de evento de la vista del primer ejemplo del uso de controles Widgets

Lo primero que ha de verificar este manipulador es el objeto que genera el evento —para la situación en la que varios objetos utilizan el mismo manipulador de evento (no es el caso)—, esto se comprueba en la línea 4 del código anterior. Este manipulador concreto, cambia el texto del *objeto MiTextoVariable*, mostrando de forma alternativa, las cadenas «Segunda cadena» y « Primera cadena » en cada pulsación del botón *MiBoton*.

En el segundo ejemplo se pretende mostrar cómo se usan los botones de exclusión mutua. Se crean tres botones de exclusión mutua que serán almacenados en un *array*, lo que permite un manejo más eficiente de los mismos. El archivo de cabecera de la vista es el siguiente:

```
1  #include <touchgfx/widgets/RadioButton.hpp>
2  #include <touchgfx/widgets/RadioButtonGroup.hpp>
3
4  class MiVista : public View<FftPresenter>
5  {
6   public:
7     MiVista(){}
8     void setupScreen();
9
10  private:
11    static const uint16_t NUMERO_BOTONES_EXCLUYENTES = 3;
12    RadioButtonGroup<NUMERO_BOTONES_VENTANAS> grupoBotonesExcluyentes;
13    RadioButton botonesExcluyentes[NUMERO_BOTONES_EXCLUYENTES];
14 }
```

Código 1.42: Cabecera de la vista del segundo ejemplo de controles Widgets

Como en el primer ejemplo, lo primero a incluir en el archivo de cabecera de la vista, son todas aquellas cabeceras necesarias para poder usar los *Widgets* requeridos —líneas 1 y 2—. En este caso no es necesario construir ningún componente en el constructor de la vista. En la parte pública solo aparece el método *setupScreen*, ya que para este ejemplo no se va a utilizar manipulador de evento de usuario. En la parte privada aparece una variable estática constante —línea 11— que fija el número de botones excluyentes dentro del *array* —línea 13—. Para que estos tres botones sean mutuamente excluyentes, han de pertenecer a un mismo grupo —línea 12—. Este grupo se trata de una clase de tipo plantilla, donde se debe pasar el número de botones que forman parte del grupo. La concreción de los botones que pertenecen al grupo, así como la configuración de cada uno de ellos, se lleva a cabo en la definición del método *setupScreen* del archivo fuente de la vista:

```
1  #include <gui/Mi_screen/MiVista.hpp>
2  #include <texts/TextKeysAndLanguages.hpp>
3  #include <BitmapDatabase.hpp>
4
5  void MiVista::setupScreen()
6  {
7   for (int i = 0; i < NUMERO_BOTONES_EXCLUYENTES; i++)
8   {
9       grupoBotonesExcluyentes.add(botonesExcluyentes[i]);
10      botonesExcluyentes[i].setBitmaps(Bitmap(BITMAP_BOT_OP_DES_ID),
11                                       Bitmap(BITMAP_BOT_OP_DES_PRES_ID),
12                                        Bitmap(BITMAP_BOT_OP_SEL_ID),
13                                       Bitmap(BITMAP_BOT_OP_SEL__PRES_ID));
14
15  botonesExcluyentes[i].setXY(HAL::DISPLAY_WIDTH/4*(i+1)-botonesExcluyentes[i].getWidth()/2,
16                              HAL::DISPLAY_HEIGHT/2 - botonesExcluyentes[i].getHeight()/2);
17  }
18
19  botonesExcluyentes[1].setSelected(true);
20
21  for (int i = 0; i < NUMERO_BOTONES_EXCLUYENTES; i++)
22    add(botonesExcluyentes);
23 }
```

Código 1.43: Función setupScreen de la vista del segundo ejemplo del uso de controles Widgets

Las inclusiones que aparecen entre las líneas 1 a 3 son las ya comentadas en el primer ejemplo. Dentro del bucle de tipo *for*, que está entre las líneas 7 a la 17, se incluye cada botón excluyente al grupo *grupoBotonesExcluyentes* —línea 9—, se especifica los bitmaps a usar en cada estado de cada uno de los tres botones —líneas de la 10 a la 13— y se posicionan —líneas 15 y 16—. Luego, se selecciona como activo el segundo botón del array —línea 16—. Finalmente, cada uno de los botones de exclusión mutua son añadidos a la vista —bucle for entre la líneas 21 y 22—.

1.14.1.2 Containers

Los contenedores, como se describió en la sección 1.6, son componentes que albergan otros componentes, de hecho, la vista es en sí misma un contenedor. No tienen una apariencia gráfica definida salvo la que aportan los controles que contiene. Si se desea que tenga una apariencia, sobre la que «descansen» el resto de componentes contenidos, se ha de añadir, como primer componente, uno de tipo *Box*, que dé al contenedor un color —sólido o semitransparente—, o de tipo *Image*, que establezca la imagen de fondo. Lo tipos de contenedores disponibles en la versión de *TouchGFX* utilizada son:

- Container
- ListLayout
- ScrollableContainer
- ZoomAnimationImage

La clase *Container* ya ha sido comentada —sección 1.6—. *ListLayout*, por su parte, representa un contenedor en el que los componentes se van posicionando de forma adyacente al ser añadidos. Esta «colocación automática» se puede realizar en las direcciones de arriba hacia abajo —SOUTH—, de abajo hacia arriba —NORTH—, de derecha a izquierda —WEST— y de izquierda a derecha —EAST—. La dirección por defecto es SOUTH. Para cambiar la dirección, en la que los componentes son añadidos, hay que pasarle al constructor de este tipo de contenedor, la dirección deseada. Es por ello que si no se quiere usar la dirección por defecto, se ha

de especificar la nueva dirección en su constructor, llamado desde la lista de inicialización del constructor de la vista. Este contenedor ha sido utilizado en este proyecto para albergar los botones de estado —sección 4.4.1.8—, o para desarrollar el componente *Selector* — sección 4.4.1.5—. El contenedor *ScrollableContainer*, permite desplazar el contenido del mismo —los componentes contenidos— si este está fuera de los límites del mismo. Habitualmente se usa conjuntamente con *ListLayout*. En este proyecto se ha usado para el desarrollo del componente *Selector* y *Menu* —sección 4.4.1.10—. Finalmente está *ZoomAnimationImage*, que está especializado en la animación de ampliación y reducción de imágenes. Este último contenedor no ha sido utilizado en el proyecto. Para clarificar el uso de contenedores, se muestra el siguiente ejemplo donde se emplean los contenedores:

```
1  #include <touchgfx/widgets/Button.hpp>
2  #include <touchgfx/containers/ListLayout.hpp>
3  #include <touchgfx/containers/ScrollableContainer.hpp>
4  #include <touchgfx/Callback.hpp>
5
6  class MiVista : public View<FftPresenter>
7  {
8   public:
9     MiVista():BotonCallback(this, &MiVista::Al_Boton),lista(touchgfx::EAST){}
10    void setupScreen();
11
12  private:
13   #define N_BOTONES 4
14   Button Botones[N_BOTONES];
15   ListLayout lista;
16   ScrollableContainer contenedoDesplazable;
17   Callback<MiVista, const AbstractButton &> BotonCallback;
18 }
```

Código 1.44: Cabecera de la vista ejemplo del uso de contenedores

Se trata de la cabecera de una vista donde se utilizan los contenedores de tipo *ListLayout* y *ScrollableContanier* de forma conjunta. Se pretende realizar una organización de botones, en un área de la pantalla, en la que necesariamente no quepan todos, sino que se puedan desplazar estos botones dentro de esta área —prácticamente igual a lo realizado en el proyecto con los botones de estado, ver sección 4.4.1.8—. En el constructor de la vista se llama a los constructores del objeto de tipo *CallBack* —*BotonCallback*—, que establece el manipulador de evento para los botones —método *Al_Boton*— en la línea 9, y del objeto *ListLayout* —*lista*—, para indicar que la dirección en la que se añaden los componente a este contenedor es EAST. Los botones son declarados como un array —línea 14— de dimensión 4 —establecida por la constate del preprocesador que aparece en la línea 13—. Seguidamente se declaran los contenedores *lista* y *contenedorDesplazable*, y finalmente el objeto *BotonCallback* de tipo *Callback*. En el archivo fuente de esta vista, se definen el método de configuración de controles, *setupScreen*, y el manipulador de eventos de los botones, *Al_Boton* :

```
1  #include <gui/Mi_screen/MiVista.hpp>
2  #include <texts/TextKeysAndLanguages.hpp>
3  #include <BitmapDatabase.hpp>
4
5  void MiVista::setupScreen()
6  {
7   for(uint8_t i=0; i<N_BOTONES; i++)
8    {
9          Botones[i].setBitmaps(Bitmap(BITMAP_BOTON_ID), Bitmap(BITMAP_BOTONPULSADO_ID));
10         Botones[i].setCallBack(BotonEstadosCallback);
11         lista.add(Botones[i]);
12   }
13   contenedoDesplazable.add(lista);
14
15   add(contenedoDesplazable);
16 }
17
18 void Al_Boton(const AbstractButton &button)
19 {
20         if(&button == &Botones[0])
21                 //código para el primer botón
22         if(&button == &Botones[1])
23                 //código para el segundo botón
24         if(&button == &Botones[2])
25                 //código para el tercer botón
26         if(&button == &Botones[3])
27                 //código para el cuarto botón
28 }
```

Código 1.45: Código fuente de la vista ejemplo del uso de contenedores

En el bucle de tipo *for* —líneas de la 7 a la 12— del método *setupScreen*, se configura la posición y se establecen los bitmaps de cada botón. En este mismo bucle son añadidos al contenedor de tipo *ListLayout* —*lista*—, que hace que cada botón aparezca junto al anterior añadido al contenedor, de tal forma que queden ordenados, gráficamente, de izquierda a derecha. Como es posible que algún botón no quede visible, debido a que la suma de sus anchuras sea mayor a la anchura del contenedor *lista*, todo este contenedor es añadido dentro del contenedor *contenedorDesplazable* de tipo *ScrollableCotainer* —línea 13—, lo que hace que el contenido del contendor se pueda desplazar al realizar un arrastre —evento *Drag*— sobre el mismo, permitiendo acceder a todos los botones. Finalmente en el método manipulador de eventos, Al_Boton —líneas de la 18 a la 28—, se identifica el botón que genera el evento, dentro de cada bloque condicional correspondiente, para dar una respuesta adecuada a cada botón.

1.14.1.3 Mixins

Las clases *Mixins* son un tipo de plantillas que dan una funcionalidad concreta a otras clases que no tienen tal funcionalidad como puedan ser las clases tipo Box o Image. También pueden constituir un método para crear nuevas clases. Las clases Mixins disponibles en la versión de TouchGFX utilizada son:

- ClickListener
- Draggable
- FadeAnimator
- MoveAnimator
- PreRenderable
- Snapper

La clase *ClickListener* permite a la clase con la que se «mezcla», gestionar el evento *ClickEvent*. La clase *Draggable* permite, a la clase al que se le aplica, moverla por la pantalla al pulsar sobre el objeto y arrastrarlo sobre la misma. *FadeAnimator* hace que el objeto al que se le aplica, aparezca o desaparezca de forma paulatina. *MoveAnimator* permite mover un objeto desde la posición actual a una posición final especificada. Esta animación es llevada a cabo mediante una ecuación representada por la clase *EasingEquations* —ver sección 1.15—. Finalmente está *Snapper*, similar a *Draggable* con la diferencia que se puede especificar una posición de enclavamiento cuando el arrastre ha finalizado. En este proyecto solo se ha utilizado el *Mixin ClickListener*, aplicado a un objeto de tipo Marcador —clase creada para este proyecto para señalizar un lugar en pantalla mediante parpadeo, ver sección 4.4.1.3—.

Como ejemplo de utilización de una clase *Mixin*, se presenta la siguiente vista:

```
1  #include <touchgfx/widgets/Box.hpp>
2  #include <touchgfx/mixins/Draggable.hpp>
3  #include <touchgfx/widgets/Image.hpp>
4  #include <touchgfx/Color.hpp>
5
6  class MiVista : public View<FftPresenter>
7  {
8   public:
9     MiVista(){}
10    void setupScreen();
11
12  private:
13    Box Fondo;
14    Draggable <Image> Icono;
15 }
```

Código 1.46: Cabecera de la vista ejemplo del uso de Mixins

Se trata del archivo de cabecera de una vista, donde se pretende que una imagen, a modo de icono en la pantalla, pueda ser movido cuando se pulsa sobre él y se arrastra por la pantalla. Para ello se declara un control de tipo *Box* —*Fondo*— que creará un fondo de color en toda la pantalla y una imagen que servirá de icono. Es de destacar la sintaxis de la creación de esta imagen con la aplicación del *Mixin Draggable* —línea 14—. Esta sintaxis es la típica de una plantilla en c++, lo que se declara es un objeto de tipo *Draggable* que internamente va a utilizar una clase indeterminada que se especifica en esta declaración, *Image*, entre los operadores de menor que y mayor que. Lo que hace internamente este *Mixin* es gestionar el evento *Click* y *Drag* para mover y arrastrar el objeto de la clase pasada a la plantilla —para ver un ejemplo de funcionamiento interno de un *Mixin* ver sección 1.13.5 o consultar el código, que está disponible—.

El archivo del código fuente de esta vista se limita a la definición de su función de configuración de controles, la función *setupScreen*:

```
1  #include <gui/Mi screen/MiVista.hpp>
2  #include <texts/TextKeysAndLanguages.hpp>
3  #include <BitmapDatabase.hpp>
4
5  void MiVista::setupScreen()
6  {
7   Fondo.setPosition (0, 0, HAL::DISPLAY_WIDTH, HAL::DISPLAY_HEIGHT);
8   Fondo.setColor(Color::getColorFrom24BitRGB(0x00, 0x00, 0xFF));
9
10  Icono.setBitmap(Bitmap(BITMAP_ICONO));
11  Icono.setXY(HAL::DISPLAY_WIDTH/2, HAL::DISPLAY_HEIGHT/2);
12  add(Fondo);
13  add(Icono);
14 }
```

Código 1.47: Código fuente de la vista ejemplo del uso de Mixins

En esta función se configura el control Fondo para ocupar toda la pantalla y que la rellene de un color azul sólido. Luego se especifica la imagen que se cargará en el objeto Icono y se colocará en el centro de pantalla. Para finalizar, ambos controles serán añadidos a la vista.

1.14.1.4 Componentes disponibles en el repositorio de TouchGFX

TouchGFX ofrece un repositorio con controles gráficos que se suministran con el código fuente. Este repositorio se encuentra en la siguiente dirección —a la fecha de creación de este documento—:

https://github.com/draupnergraphics/touchgfx-open-repository/ [2]

En él se ofrecen varios componentes de utilidad, como son: LinearGauge, ExtendedZoomAnimationImage, Carousel, Gauge, WheelSelector, CircularProgress, Lens, Graph, QRCode, Animated Gauge, DotIndicator y SwipeContainer. Por ejemplo, Gauge permite mostrar un indicador de aguja similar a un barómetro o velocímetro analógico de un coche; Lens crea un efecto lente en una región de pantalla o Graph permite crear un gráfico. No hay que confundir este último componente con el utilizado en este proyecto, que a pesar de denominarse igual, este se trata de un componente derivado de CanvasWidget —renderizado vectorial—, mientras que el creado explícitamente para este proyecto, deriva directamente de Widget —ver sección 4.4.1.11—. De todos ellos, los dos utilizados en este proyecto son: DotIndicator y SwipeContainer. Este último se trata de un contendor que puede albergar varias imágenes —han de ser de igual tamaño—, donde solo se muestra una en pantalla y para acceder a las demás, se debe arrastrar la actual —hacia la derecha o la izquierda— para que aparezca la siguiente —del mismo modo que se suelen visualizar varias fotografías en un móvil—. Para indicar al usuario cuantas fotos hay disponibles y cuál es la que se está visualizando, si utiliza el control DotIndicator. Consiste en una secuencia de puntos, tantos como fotos hay en el contendor SwipeContainer, y solo uno de ellos está iluminado: aquel que corresponde con el cardinal de la imagen actual en pantalla. Tal y como se muestra en el repositorio, el aspecto que puede tener este último control es el siguiente [2]:

Ilustración 1.35: Aspecto del control DotIndicator

En la figura se muestra este control rodeado por la elipse roja. Indica que hay cuatro fotos y la mostrada en este momento es la segunda. Para controlar este *Widget*, se dispone de varios métodos como es el dedicado a asignar los *bitmaps* de sus diferentes estados —*setBitmaps()*—, indicar cuál es el punto iluminado —*setHighlightPosition()*—, configurar el número de puntos que se van a mostrar —*setNumberOfDots()*—, o cambiar el punto iluminado al siguiente o al anterior —*goRight()* y *goLeft()*—. Sin embargo, como este control se va a utilizar en conjunción con *SwipeContainer*, será este último el que maneje todos estos métodos, y solo se

deberá especificar los bitmaps de los estados de *DotIndicator* y la alineación de los mismos respecto al contenedor *SwipeContainer* —a través de métodos de *SwipeContainer* —. Para que esta asociación funcione, el archivo de cabecera y fuente de *DotIndicator* ha de estar en el directorio gui/common/, en caso contrario, es necesario editar y cambiar el archivo de cabecera de *SwipeContainer*. Por su parte, este último control, al tratarse de un contendor, bastará con añadir las Imágenes, que se quieren que se visualicen al realizar el arrastre por parte del usuario, suministrar los bitmaps para el control *DotIndicator*, que internamente manejará el control *SwipeContainer*, e indicar la alineación de los puntos de *DotIndicator*. Un ejemplo del aspecto de *SwipeContainer*, que TouchGFX suministra en su repositorio es la siguiente [2]:

Ilustración 1.36: Aspecto del control SwipeContainer

Suponiendo un caso en el que se dispone de tres imágenes, en este repositorio se muestra el siguiente ejemplo [2]:

```
#include <gui/common/SwipeContainer.hpp>

class MiVista : public View<FftPresenter>
{
 public:
   MiVista(){}
   void setupScreen();

 private:
   SwipeContainer imageContainer;
   Image images[3];
}
```

Código 1.48: Cabecera de la vista ejemplo del uso de SwipeContainer

Se trata de un archivo de cabecera de una vista en la que se declaran un contenedor de tipo SwipeContainer —imageContainer—, y un array de tres imágenes —images—. Hay que destacar que no es necesario declarar un objeto de tipo *DotIndicator*, el objeto de tipo *SwipeContainer* contendrá uno internamente, lo que es necesario es que el archivo fuente y de cabecera de la clase *DotIndicator* esté en la ruta mencionada. El archivo fuente correspondiente de esta vista es [2]:

```
1  #include <gui/Mi_screen/MiVista.hpp>
2  #include <BitmapDatabase.hpp>
3
4  void MiVista::setupScreen()
5  {
6   images[0].setBitmap(Bitmap(BITMAP_IMAGE_00_ID));
7   images[1].setBitmap(Bitmap(BITMAP_IMAGE_01_ID));
8   images[2].setBitmap(Bitmap(BITMAP_IMAGE_02_ID));
9   imageContainer.add(images[0]);
10  imageContainer.add(images[1]);
11  imageContainer.add(images[2]);
12  imageContainer.setXY(0, 0);
13  imageContainer.setDotIndicatorBitmaps(Bitmap(BITMAP_DOT_INDICATOR_ID),
14                                        Bitmap(BITMAP_DOT_INDICATOR_HIGHLIGHTED_ID));
15  imageContainer.setDotIndicatorXYWithCenteredX(imageContainer.getWidth()/2,
16                                                imageContainer.getHeight() - 16);
17  imageContainer.setSwipeCutoff(80);
18  imageContainer.setEndSwipeElasticWidth(30);
19  add(imageContainer);
20 }
```

Código 1.49: Código fuente de la vista ejemplo del uso de SwipeContainer

Primero se añaden los bitmaps a los controles de tipo imagen —líneas de la 6 a la 8—, luego se añaden estos controles al contenedor de tipo SwipeContainer —líneas de la 9 a la 11—, seguidamente se fijan los bitmps de los indicadores de puntos y su centrado dentro del contenedor SwipeContainer —líneas de la 13 a la 16. Para finalizar, se configura cuanto se debe a arrastrar una imagen para que pase a la siguiente —línea17—, se determina cuanto se puede arrastrar las imágenes de los extremos mostrando el fondo que hay debajo y se añade el contenedor SwipeContainer a la vista —línea 19—.

En este proyecto no se ha utilizado este contenedor cargando imágenes, tal y como se ha mostrado en el ejemplo anterior; se ha utilizado cargando otros contenedores donde cada uno contiene sus propios controles así como posibles imágenes de fondo. Los menús que implementan este contenedor son: CONTROLES, PANTALLA y CURSORES —ver sección 4.1—. En el caso de PANTALLA, cada contenedor tiene su propio fondo, con lo que al arrastrar cada uno de ellos, para acceder al siguiente, se arrastra tanto los componentes contenidos en el contendor como su fondo. En cambio, en los otros dos menús, cada contenedor carece de fondo propio, con lo que al arrastrar cada uno de ellos, solo se arrastran los controles contenidos visualizando, de forma estática, el fondo que pertenece directamente al menú.

1.15 Clases de utilidad en TouchGFX

TouchGFX tiene multitud de clases de utilidad que no constituyen componentes gráficos, sin embargo, hay dos a destacar: *EasingEquations* y *GPIO*.

EasingEquations es una clase útil para realizar animaciones. Todos sus métodos públicos son estáticos —no se necesita instancia de clase para utilizarlos—. Estos métodos representan ecuaciones que transforman unos datos de entrada en unos de salida acorde con la ecuación que representa el método concreto. Su finalidad es la de crear una velocidad en la animación que no sea lineal, dándole un carácter más realista. Cada uno de estos métodos admite cuatro argumentos que son, por este orden: tiempo o paso de la animación, valor inicial de salida, diferencia entre el valor inicial y final de salida, y duración o pasos totales de la animación. Es decir, los datos de entrada —pasos de la animación— constituyen la variable independiente de la ecuación y los datos de salida, la variable dependiente. Todos ellos son de tipo entero con signo de 16 bits. La representación de gráfica de estas ecuaciones es la siguiente [23]:

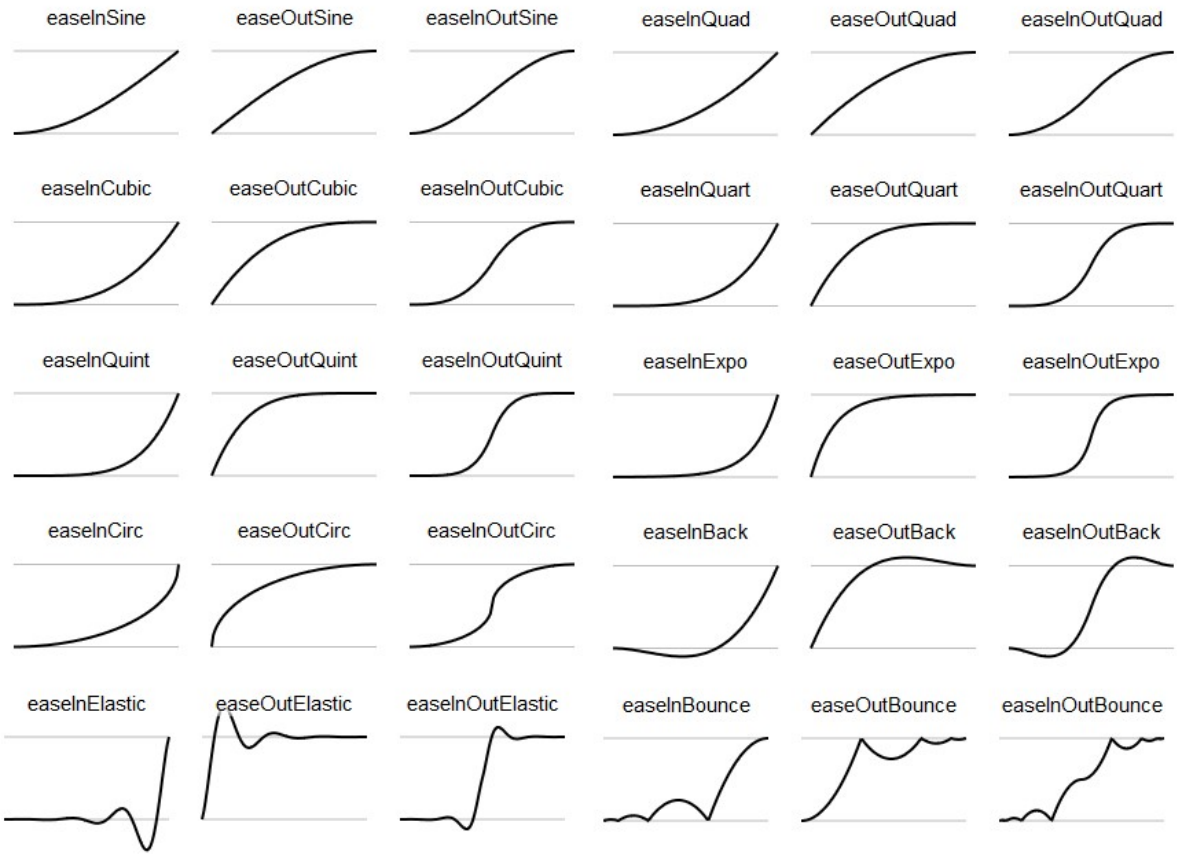


Ilustración 1.37: Representación gráfica de las ecuaciones EasingEquations

La definición pública de la clase es la siguiente:

```
1  namespace touchgfx
2  {
3   typedef int16_t (*EasingEquation)(uint16_t, int16_t, int16_t, uint16_t);
4
5
6   class EasingEquations
7   {
8    public:
9       static int16_t backEaseIn(uint16_t t, int16_t b, int16_t c, uint16_t d);
10
11      static int16_t backEaseOut(uint16_t t, int16_t b, int16_t c, uint16_t d);
12
13      static int16_t backEaseInOut(uint16_t t, int16_t b, int16_t c, uint16_t d);
14
15      . . .
16
17      static int16_t sineEaseInOut(uint16_t t, int16_t b, int16_t c, uint16_t d);
18
19   private:
20       . . .
21  };
22 }
```

Código 50: Clase EasingEquations

Cada método público representa cada una de las ecuaciones mostradas es la ilustración anterior, conteniendo los cuatro argumentos antes descritos. Cuando se realiza una animación a través de una de estas ecuaciones, el programador debe llamarlas de forma repetida en cada paso de animación —primer argumento— para obtener su valor de salida correspondiente —el valor retornado por el método—. Los tres restantes

argumentos permanecerán constantes entre llamada y llamada. Ya que todos estos métodos tienen el mismo prototipo —iguales argumentos e igual valor devuelto—, *TouchGFX* facilita un puntero a función —línea 3—, fuera de la definición de la clase, pero dentro del espacio de nombres «touchgfx», que permite manejarlas de una forma unificada y versátil para poder cambiar de ecuación pero manejar la misma variable —el puntero al método— en el código. Un código ejemplo que muestra su utilización es el siguiente:

```
1  EasingEquation Ecuacion;
2  int16_t pasos_animacion;
3  int16_t posicion_inicial;
4  int16_t posicion_final;
5  int16_t posicion_actual;
6  . . .
7  Ecuacion=&EasingEquations::quadEaseOut;
8  for(int16_t paso_animacion; paso_animacion<=pasos_animacion; paso_animacion++)
9  {
10   posicion_actual=(int16_t) Ecuacion(pasos_animacion,
11                                      posicion_inicial,
12                                      posicion_final-posicion_inicial,
13                                      pasos_animacion);
14
15   //se actualiza la posición del elemento gráfico a animar
16   //acorde con el valor de posicion_actual.
17 }
```

Código 1.51: Uso de las ecuaciones EasingEquatios

Primero se debe declarar el puntero a función de tipo *EasingEquation* —línea 1—. Este puntero generalmente es un dato miembro de una clase. Luego se declaran las variables necesarias para realizar la animación. Se necesita una variable que albergue el paso actual de animación —línea 2—, otra que contenga el valor de la posición inicial del objeto a animar —línea 3—, otra que especifique la posición final —línea 4— y otra que contenga la posición para cada paso de animación que será devuelto por la ecuación. Todas estas variables, al igual que la del putero a la ecuación, suelen ser atributos de clase, y sus valores —no mostrados en el código de ejemplo—, son cargados en el constructor de la clase y modificados mediante métodos de acceso a atributos. Luego se carga el puntero de tipo *EasingEquation* con la dirección del método concreto de la clase *EasingEquations* a utilizar —línea 7—. En el caso concreto del código mostrado, se utiliza la ecuación *quadEaseOut*. Este valor se suele cargar en el constructor de la clase donde está declarado el puntero de tipo *EasingEquation*. Finalmente se procede a determinar, para cada paso de animación, qué posición le corresponde tener al objeto a animar. Esto se hace mediante el bucle for entre las líneas 8 a la 17. Aunque para este ejemplo, la animación consiste en el movimiento de un objeto, no siempre es así. Una animación puede consistir en la desaparición de un elemento, variando su transparencia acorde con una de estas ecuaciones.

Estas ecuaciones han sido utilizadas en algunos de los nuevos componentes gráficos desarrollados —ver sección 4.4—.

La otra clase de utilidad, *GPIO*, permite que el desarrollador tenga acceso a señales de temporización de funcionamiento interno de *TouchGFX*. La definición de esta clase es la siguiente:

```
class GPIO
{
public:
  typedef enum
  {
    VSYNC_FREQ,
    RENDER_TIME,
    FRAME_RATE,
    MCU_ACTIVE
  } GPIO_ID;

   static void init();

   static void set(GPIO_ID id);

   static void clear(GPIO_ID id);

   static void toggle(GPIO_ID id);

   static bool get(GPIO_ID id);
};
```

Código 1.52: Definición de la clase GPIO

Contiene la definición de un tipo enumerado —*GPIO_ID*— que indica que las posibles señales a las que se tiene acceso son: el sincronismo vertical —*VSYNC_FREQ*—, el pin configurado conmuta en cada sincronismo; el tiempo que se está consumiendo para la representación en el buffer secundario —*RENDER_TIME*—, la conmutación de buffers —*FRAME_RATE*—, y el tiempo que está activo el núcleo del micro —*MCU_ACTIVE*—. Mediante el método *init*, *TouchGFX* inicializa los pines PG.2, PG.9, PG.13 y PG.14 para ser las señales, respectivas, *VSYNC_FREQ*, *RENDER_TIME*, *FRAME_RATE* y *MCU_ACTIVE*. *TouchGFX* utiliza los métodos *set*, *clear* y *toggle* para actualizar estas señales. Sin embargo, estos métodos son públicos y el desarrollador tiene acceso a ellos. Por ejemplo, se podría utilizar el método *get* de esta clase, para dar salida a cualquiera de estas señales por otro pin de cualquier puerto.

2 Sistema operativo en tiempo real FreeRTOS

2.1 Introducción

En este capítulo se va a exponer una breve introducción del funcionamiento de un sistema operativo en tiempo real para el manejo de tareas concurrentes: *FreeRTOS* V7.6.0. Este conocimiento se hace necesario a la hora de desarrollar una aplicación en *TouchGFX*, debido a que ya trae consigo este sistema operativo para el manejo del funcionamiento de la interfaz gráfica. Además, debido a que la función encargada de realizar la representación gráfica está encapsulada en una tarea que constituye una ejecución en un bucle infinito, y no se dispone del código fuente de la mayor parte de esta librería *TouchGFX*, se hace necesario el uso de otras tareas para desarrollar la aplicación en sí. El hecho del uso de *FreeRTOS* como sistema operativo a utilizar, es debido a que se trata de un sistema totalmente gratuito, se dispone de su código fuente, se puede utilizar libremente en aplicaciones comerciales, no se tiene que facilitar el código del desarrollo creado con él y viene implementado en *TouchGFX*. En esta sección también se explicará el comportamiento de *FreeRTOS* respecto a interrupciones hardware, el modelo de memoria que usa dentro de *TouchGFX* y como se configura el sistema operativo.

2.2 Descripción del funcionamiento del sistema operativo FreeRTOS.

FreeRTOS es un sistema operativo en tiempo real. Esto significa que dispondrá de un elemento llamado planificador —scheduler— que multiplexará la ejecución de las tareas disponibles en el sistema y lo hará proporcionando un patrón de ejecución predecible, lo que es conocido como patrón determinista [10]. Esto significa que una vez diseñada todas las tareas con sus prioridades correspondientes, será posible determinar que cierta tarea pueda responder ante determinado evento dentro de un tiempo definido.

El planificador se encarga de decidir que tarea ha de ocupar el corazón de la CPU en cada momento y por tanto cual es la tarea que realmente se está ejecutando. El reparto de la ejecución que hace el planificador en el tiempo, da la «ilusión» de ejecución concurrente. En *FreeRTOS* —y en general en cualquier sistema operativo de tiempo real—, una tarea solo puede estar en cuatro estados que son: ejecución, bloqueado, suspendido y preparado. El siguiente esquema muestra dichos estados y las transiciones entre ellos [10]:

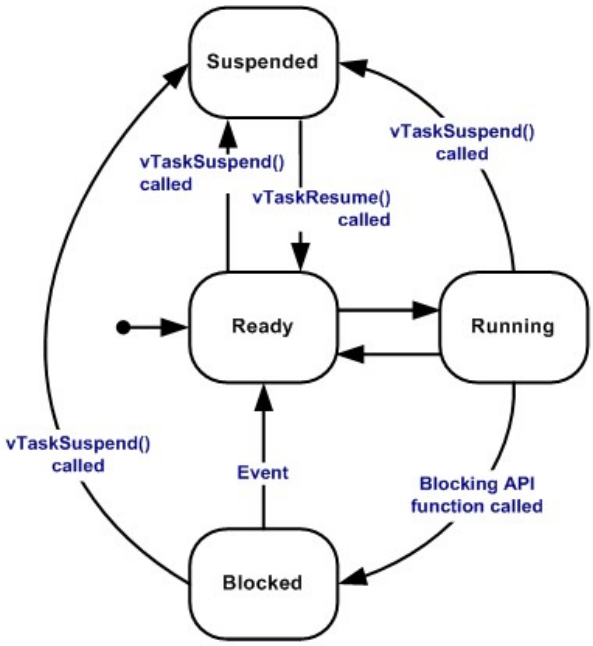


Ilustración 2.1: Esquema del funcionamiento de FreeRTOS

Una tarea está en estado de ejecución cuando está ocupando el tiempo de procesamiento del corazón de la *CPU*. Se encuentra en estado preparado cuando aún estando en condición de ser ejecutada, no está aún ocupando el corazón de la *CPU* debido a que hay otra tarea de igual o mayor prioridad ejecutándose. En este sentido, se puede decir que la tarea está a la espera de que el corazón esté libre para ser ejecutada. Una tarea entra en este estado por la decisión del planificador. En estado bloqueado se encuentra toda tarea que esté a la espera de cierto evento externo o periodo de tiempo para poderse encontrar en condiciones de ser ejecutada. Generalmente la tarea entra en este estado por decisión propia, como pueda ser la situación en la que espera a que esté libre el testigo de determinado semáforo, o cuando espera durante un tiempo para volver a ejecutarse. Por último, está el estado de suspensión. Una tarea entra en este estado mediante la orden de suspensión —generalmente enviada por otra tarea— y no podrá salir del mismo hasta que haya una orden explicita de reanudación —por parte de otra tarea—.

2.3 Breve descripción de la utilización de FreeRTOS

La utilización de *FreeRTOS* no es el objeto de este proyecto, pero, debido a que es el sistema operativo que viene por defecto con *TouchGFX*, además, necesita de un sistema operativo para funcionar —aunque puede funcionar de una forma rudimentaria sin sistema operativo—, y el método de la capa *HAL* que realiza el trabajo del entrono gráfico, ha de estar encapsulado en una tarea, se hace necesario el conocimiento del manejo de tareas, colas y semáforos.

Las tareas constituyen cada uno de los hilos de ejecución que van a ser ejecutados de forma concurrente —se multiplexará su ejecución con el resto de tareas en el sistema—. Las tareas ejecutarán funciones con el prototipo *void [nombre_función](void * pvParameters)* y se crean mediante la siguiente función de *FreeRTOS*:

```
1  BaseType_t xTaskCreate(
2                            TaskFunction_t pvTaskCode,
3                            const char * const pcName,
4                            unsigned short usStackDepth,
5                            void *pvParameters,
6                            UBaseType_t uxPriority,
7                            TaskHandle_t *pvCreatedTask
8                          );
```

Código 2.1: Función de FreeRTOS para crear una tarea

El primer parámetro —línea 2— es un puntero a la función que se quiere ejecutar de forma concurrente y que constituirá la tarea. Esta función no debe retorna jamás, es decir, su ejecución a de ir dentro de un bucle infinito. El segundo, —línea 3— es una cadena de texto que da un nombre descriptivo a la tarea, su longitud está limitada a la definición «*configMAX_TASK_NAME_LEN*» del archivo de configuración del sistema «*FreeRTOSConfig.h*» —la configuración se comenta en la sección "Configuración del sistema operativo en TouchGFX" de este mismo capítulo—. El tercer argumento hace referencia al tamaño de la pila utilizada por la tarea. La pila de la tarea es un área de memoria creada para manejar los datos de la propia tarea. El sistema la alberga en el área libre —*heap*— y es una memoria de tipo «*filo*» —también conocida como *lifo*: último en entrar, primero en salir—. Se utiliza para alojar las variables internas de la tarea, las llamadas a las funciones, consistentes en la dirección de retorno de la función —punteros que albergan datos de 32 bits— y sus argumentos, así como para alojar las variables internas de las funciones llamadas [10]. Todo ello habrá que tenerlo en cuenta para establecer el tamaño de la pila para la tarea, recomendando redimensionar la misma con 512 bytes extra [10]. El tamaño de la pila a poner en este tercer parámetro está dado en elementos cuyo ancho es el que maneja el micro concreto –en este caso 4 bytes–. Así mismo, la estimación del tamaño que representan todas las tareas ha de tenerse en cuenta a la hora de concretar la definición «*configTOTAL_HEAP_SIZE*» del archivo de configuración, que *FreeRTOS* utiliza para reservar el array que representará la memoria disponible para todas las tareas y demás elementos del sistema. El cuarto parámetro —línea 5— constituye el argumento a ser enviado a la función que actuará como tarea. El quinto parámetro —línea 6— es la prioridad de la tarea. Este valor determina qué tarea tiene mayor preferencia de ejecución sobre el resto, a mayor valor de prioridad, mayor preferencia de ejecución. El último parámetro —línea 7— es el manejador de la tarea. Este valor es devuelto por la función «*xTaskCreate*» y será utilizado por el sistema para hacer referencia a la misma después de su creación. Si después de su creación no se va a hacer referencia a la tarea desde ninguna otra función del sistema operativo, se puede dejar este valor a cero. La función «*xTaskCreate*» devuelve el valor «*pdPASS*» si ha creado y alojado en memoria correctamente la tarea, sino, devuelve algún código de error definido en el archivo «*projdef.h*».

El siguiente fragmento de código muestra la creación de la tarea «*gui*» —entorno gráfico— utilizada por *TouchGFX*:

```
1  /**
2   * Define the FreeRTOS task priorities and stack sizes
3   */
4  #define configGUI_TASK_PRIORITY                 ( tskIDLE_PRIORITY + 3 )
5
6  #define configGUI_TASK_STK_SIZE                 ( 950 )
7
8
9  static void GUITask(void* params)
10 {
11   touchgfx::HAL::getInstance()->taskEntry();
12 }
13
14
15 int main (void)
16 {
17   transfiereA_RAM();
18         hw_init();
19   touchgfx_init();
20
21
22   /**
23    * IMPORTANT NOTICE!
24    *
25    * The GUI task stack size might need to be increased if creating a new application.
26    * The current value of 950 in this template is enough for the examples distributed with
27    * TouchGFX, but is often insufficient for larger applications. For reference, the
28    * touchgfx_2014 demo uses a stacksize of 2000.
29    * If you experience inexplicable hard faults with your custom application, try
30    * increasing the  stack size.
31    * Also note that in FreeRTOS, stack size is in words. So a stack size of 950 actually
32    * consumes  3800 bytes of memory. The configTOTAL HEAP SIZE define in
33    * platform/os/FreeRTOSConfig.h configures the total amount of memory available for
34    * FreeRTOS, including task stacks. This value is expressed in BYTES, not words. So,
35    * with a stack size of 950, your total heap should be at least 4K.
36    */
37   xTaskCreate( GUITask, (signed char*)"GUITask",
38                configGUI_TASK_STK_SIZE,
39                NULL,
40                configGUI_TASK_PRIORITY,
41                NULL);
42
43   vTaskStartScheduler();
44
45   for(;;);
46
47 }
```

Código 2.2: Creación de tareas en FreeRTOS

En la línea 4 se define la prioridad de la tarea *GUITask* como tres niveles superior a la de la tarea por defecto «*Idle*» —que tiene la prioridad más baja—. En la línea 6 se define el tamaño de la pila para la tarea, especificada en palabras —4 bytes—. Entre las líneas 9 a 12 está la definición de la tarea *GuiTask*, consistente en la llamada al método *taskEntry* del objeto *singleton* de la clase *HAL*, que constituye la función que *TouchGFX* implementa para el funcionamiento del entorno gráfico. Entre las líneas 37 a 41 se produce la creación de la tarea *Guitask*, que no tendrá manejador de tarea, ya que no se va destruir, a no ser en la finalización de la aplicación. Finalmente está, en la línea 43, la llamada a la función *vTaskStartScheduler*, que inicia el sistema operativo. El bucle infinito que está en la línea 45 no se ejecuta ya que la tarea *GUITask* ya constituye un bucle infinito.

Para borrar tareas está la función del sistema *vTaskDelete*, cuyo único parámetro es el manejador de la tarea a ser borrada y que debe haber sido establecido en la creación de la misma. Esta función no devuelve ningún valor y para poder ser usada se ha de incluir la definición «*INCLUDE_vTaskDelete*» con valor «*1*» dentro del archivo de configuración y utilizar un modelo de memoria distinto a «*head_1.c*» —mirar la sección «Modelo de memoria que TouchGFX usa para FreeRTOS», en este mismo capítulo—.

Para el control de tareas existen varias funciones, las más importantes son *vTaskSuspend* y *vTaskResume*, que suspenden y reanudan, respectivamente, la ejecución de la tarea concreta, especificada por su manejador como argumento en dichas funciones. Para poder utilizar estas dos funciones es necesario incluir la definición *INCLUDE_VTaskSuspend* en el archivo de configuración. Otras dos funciones para el control de tareas son *vTaskDelay* y *vTaskDelayUntil*. La primera de ellas bloquea la ejecución de la tarea que la llama hasta el periodo de tiempo determinado en su argumento y especificado en tick del sistema. La segunda tiene dos argumentos, el primero de ellos es un puntero en el que se ha de especificar la cuenta de *ticks* actual, y en el segundo, el periodo en el que se quiere de se desbloquee la tarea. En la primera llamada a esta función, su primer argumento será realmente la cuenta actual de *tick*, que podrá ser obtenida mediante la función *xTasGetTickCount*, pero en sucesivas llamadas, el puntero, del primer argumento, ya contendrá la cuenta de *ticks* anterior en la que se desbloqueó la tarea. Esto hace que esta segunda función asegure el desbloqueo de la tarea en el periodo exacto, mientras que la primea solo asegura el desbloque de la tarea en el periodo de tiempo especificado que puede ser algo diferente al esperado entre llamadas a la misma. Para el uso de *vTaskDelay* se ha de especificar la declaración «*INCLUDE_vTaskDelay*» con valor «*1*» en el archivo de configuración, mientras que para *vTaskDelayUntil* se ha de incluir la declaración «*INCLUDE_vTaskDelayUntil*» con valor «*1*». Como ejemplo de uso de *vTaskDelayUntil*, se presenta el siguiente código [10]:

```
1  void vTaskFunction( void * pvParameters )
2   {
3   TickType_t xLastWakeTime;
4   const TickType_t xFrequency = 10;
5
6       // Se inicializa la variable xLastWakeTime con los ticks actuales.
7       xLastWakeTime = xTaskGetTickCount();
8
9       for( ;; )
10      {
11          // Espera hasta el siguiente periodo (especificado en xFrecuency).
12          vTaskDelayUntil( &xLastWakeTime, xFrequency );
13          // En sucesivas llamadas a vTaskDelayUntil,
14          // LastWakeTime ya tiene la cuenta actual de ticks.
15
16
17          // Resto de código.
18      }
19  }
```

Código 2.3: Utilización de temporizaciones en FreeRTOS

Las tareas no se pueden comunicar entre sí de igual modo que los hacen las funciones. Una función envía a otra función información mediante el paso de argumentos en su llamada. Esto implica que la función que envía información «ejecuta» la función que recibe dicha información. Este mecanismo no es posible con las tareas, ya que solo habrá una tarea en ejecución y el único agente que puede pasar una tarea de «preparada» a «ejecución» es el planificador. Una tarea puede crear otra tarea, pero no puede ponerla en ejecución. Es por ello que se ha de proveer un elemento que persista después de que la tarea haya pasado del estado de ejecución a cualquier otro —incluso si es destruida—, donde se aloje la información que una tarea quiere pasar a otra. Este elemento es la cola y constituye uno más del sistema operativo, en este sentido no es propiedad de ninguna tarea. Se trata de una región de memoria que el sistema crea en el área libre y es de tipo *FIFO* —el primero en entrar es el primero en salir—, aunque el sistema dispone de ciertas funciones que permiten que el comportamiento de la cola no sea estrictamente de *FIFO*. La cola puede ser de cualquier tamaño y sus elementos de cualquier tipo, de cualquier forma, estos dos parámetros habrán de tenerse en cuenta a la hora de ajustarse a

las restricciones del área de memoria máxima que *FreeRTOS* dispondrá para crear elementos del sistema operativo, como son las tareas, colas y semáforos —ver el apartado «Modelo de memoria que *TouchGFX* usa para *FreeRTOS*» en este mismo capítulo—. Para enviar un elemento al final de la cola, *FreeRTOS* dispone de la función *xQueueSend* y para recibir el elemento en el extremo final de la cola, la función *xQueueReceive*. La siguiente figura muestra la secuencia de envío de dos datos, por parte de la tarea «*Task A*» y la recepción de estos dos datos por parte de la tarea «*Task B*», todo ello a través de una cola de longitud de 5 elementos [10]:

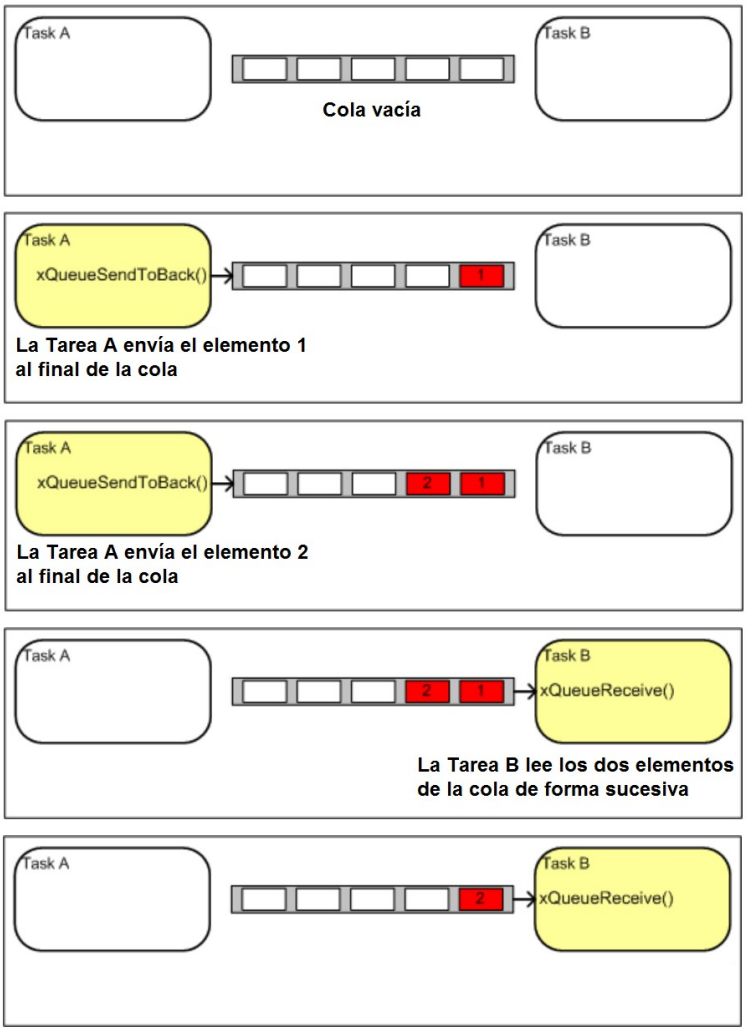


Ilustración 2.2: Ejemplo de envío de datos por una cola del RTOS

En la figura se muestra que la tarea «A» envía 2 elementos al final de la cola. Para ello se utiliza la función *xQueueSendToBack* que es realmente un sinónimo de *xQueueSend*, ya que se dispone de la función *xQueueSendToFront* que permite enviar elementos al inicio de la cola —lo que posibilita un comportamiento de

la cola no estricto de tipo *FIFO*, como se comentó antes—. La tarea *B* necesita de dos llamadas consecutivas a la función *xQueueReceive* para poder recuperar los elementos de la cola.

Para asegurar la integridad de los datos en la cola, estos son pasados por valor. De esta forma, por ejemplo, una tarea puede enviar un dato a la cola y luego ser destruida; el valor contenido en la cola «persistirá» a la existencia o estado de la tarea que lo envió.

Es posible enviar la información por referencia. En este caso los elementos de la cola han de ser punteros y por tanto, el contenido de los datos está realmente fuera de la cola. Esto es útil cuando los elementos a enviar son excesivamente grandes, como puedan ser estructuras o arrays de gran dimensión. En este caso, se ha de tener cuidado y mantener el contenido de estas estructuras y arrays hasta que sean accedidos por la tarea destino. Se ha de tener en mente, que aunque los datos se pasan por referencia, los punteros se pasan por valor, es decir, que lo que se asegura en la cola son las direcciones de los datos a ser accedidos.

Las colas se crean mediante la función *xQueueCreate* cuyos argumentos son, respectivamente, la longitud de la cola y el tamaño de cada elemento de la misma. Esta función devuelve un dato de tipo *QueueHandle_t* que constituye el manejador de la cola para poder hacer referencia a ella en otras funciones del sistema operativo.

La función de envío de información *xQueueSend* tiene tres argumentos, que son de forma respectiva, el manejador de la cola a la que enviar el dato, la dirección del dato a ser enviado —aunque se especifica la dirección del dato, el dato es enviado por valor— y el tiempo de espera, donde la tarea se bloquea en el caso de estar la cola llena. Este tiempo se especifica en *ticks*,. Si este último parámetro tiene valor cero, la tarea no se bloqueará, mientras que si su valor es «*portMAX_DELAY*», esperará de forma indefinida hasta que la cola esté vacía —en este último caso ha de estar definida la constante «*INCLUDE_vTaskSuspend*» en el archivo de configuración «*FreeRTOSConfig.h»*—. La función retorna el valor «*pdTRUE*» si ha podido enviar el dato a la cola o «*errQUEUE_FULL*» en caso negativo. Si se quiere enviar un dato a la cola desde una interrupción que controle el sistema operativo, se ha de utilizar la función *xQueueSendFromISR* —ver el apartado «Directivas de manejo de prioridades en interrupciones en el archivo FreeRTOSConfig.h» en este mismo capítulo—.

La función de recepción *xQueueReceive* tiene tres argumentos que son: el manejador de la cola a la que extraer el dato, la dirección de la variable en la que almacenar este dato y el tiempo que se bloquea la función a la espera que haya un dato disponible en la cola. Al igual que ocurre con la función de envío de datos, si este último argumento es cero, la tarea no se bloqueará, mientras que si su valor es «*portMAX_DELAY*» se quedará bloqueada de forma indefinida hasta que haya un dato en la cola. Devuelve el valor «*pdTRUE*» si se ha recibido un dato de la cola o «*pdFALSE*» en caso contrario.

Un ejemplo de utilización de tareas y colas es mostrado en la siguiente porción de código:

```
1  ////////////////////////////////// Definición de tipos ////////////////////////////////////////
2  typedef enum  TEvento{PRESIONADO, LIBERADO} Evento;
3
4   typedef struct TDATO
5   {
6      Evento eventoID;
7      short x;
8      short y;
9   } DATO;
10
11  ////////////// Definición de elementos del sistema operativo (tareas y colas) ///////////////
12  // Declaración del manejador de la cola.
13  QueueHandle_t cola;
14
15  // Definición de la tarea productora.
16  void TareaProductor( void *pvParameters )
17  {
18    // variables utilizadas por la tarea productora
19    DATO dato;
20    Evento eventoID;
21    short x, y;
22    BaseType_t estadoCola;
23
24    while(1)
25    {
26      // ObtenEvento es la función hipotética que recoge
27      // las coordenadas de pantalla pulsadas o liberadas
28      if(ObtenEvento(&eventoID, &x, &y))
29       {
30        dato.eventoID = eventoID;
31        dato.x = x;
32        dato.y = y;
33        //se envía el dato a la cola esperando 5ms a lo sumo para que esté vacía
34        estadoCola=xQueueSend( cola, ( void * ) &dato, ( TickType_t ) portTICK_PERIOD_MS(5));
35        if(estadoCola == pdTRUE)
36          printf("Se ha podido enviar el dato");
37        else
38          printf("No se ha podido enviar el dato en el intervalo de espera de 5ms");
39       }
40    }
41  }
42
43  // definición de la tarea consumidora
44  void TareaConsumidor( void *pvParameters )
45  {
46    DATO datoRecibido;
47
48     while(1)
49     {
50      // Recibe el mensaje de la cola.  Se bloquea durante 10 ms si el mensaje no ha llegado
51      // antes de ese tiempo.
52      if( xQueueReceive( cola, &datoRecibido, ( TickType_t )portTICK_PERIOD_MS(10) ) == pdTRUE)
53        printf("Se ha recibido un evento");
54      else
55        printf("No se ha recibido un evento de pantalla en los últimos 10 ms")
56     }
57  }
58
59  ////////////////////////////////////// función principal //////////////////////////////////////
60  int main (void)
61 {
62    // Creación de la cola
63    cola = xQueueCreate( 4, sizeof(DATO));
64    if( cola == 0 )
65     {
66      // código para responder al error al no poder crear la cola.
67     }
68    //Creación de las tareas
69    xTaskCreate( TareaProductor, (signed char*)"Productor",
70                 STACK_BYTES(2000),
71                 NULL,
72                 2,
73                 NULL);
74
75    xTaskCreate( TareaConsumidor, (signed char*)"Consumidor",
76                 STACK_BYTES(2000),
77                 NULL,
78                 2,
79                 NULL);
80
81    //Se inicia el planificador
82    vTaskStartScheduler();
83    while(1);
84 }
```

Código 2.4: Ejemplo de comunicación entre dos tareas RTOS

Entre las líneas de código del 2 al 9 mostradas anteriormente, se define un tipo enumerado y una estructura —que utiliza el tipo enumerado— que va a servir como tipo de dato de la cola. En la línea 13 se declara el manejador de la cola y en la línea 63 —dentro de la función *main*— se crea la misma. Entre las líneas 16 a 41 se define la primera tarea encargada de introducir elementos en la cola. Dentro de la definición de esta tarea, entre las líneas 19 a 22, se declaran variables de uso interno. A partir de la línea 24 se inicia la tarea propiamente dicha, donde se llama a la función *ObtenEvento*, función que hipotéticamente determina si ha habido una pulsación o liberación en una zona concreta de la pantalla, informando de ello en su valor de retorno, y acorde a esta situación, rellena las variables pasadas por referencia a sus argumentos. Luego, en la línea 34 se llama a la función *xQueueSend*, donde se especifica el manejador de cola a utilizar, la dirección de la variable «*dato*», que contiene el dato —estructura— a ser enviado y el periodo de tiempo que está dispuesta la tarea a esperar antes de continuar ejecutándose. Aquí se ha utilizado una macro función *portTICK_PERIOD_MS* que tiene *FreeRTOS* para especificar el tiempo de espera en milisegundos en vez de en *ticks*. El resultado de la consecución de la introducción del dato en la cola es almacenado en la variable «*estadoCola*» y verificado en el bloque «*if-else*» que se encuentra entre las líneas 35 a 38, informando si se ha podido o no enviar el dato. Entre las líneas 44 a 57 está la definición de la segunda tarea encargada de recoger los datos que envíe la primera tarea. En la línea 52 se llama a la función *xQueueRecive* con los argumentos del manejador de la cola a utilizar, la dirección de la variable en la que almacenar el dato extraído de la cola y el tiempo, en milisegundos, que la tarea se bloqueará a la espera de un nuevo dato. La función se encuentra dentro de la condición de una cláusula «*if*», debido a que su valor de retorno informa si dentro del tiempo de espera se ha obtenido o no un dato nuevo. De este hecho se informa en la línea 53 o en la línea 55. Entre las líneas 69 a 79, dentro de la función «*main*», está la creación de las dos tareas donde se ha utilizado la macro función de *FreeRTOS* «*STACK_BYTES*» para especificar el tamaño de la cola en bytes y no en el ancho del dato que maneja el procesador —en este caso de 32 bits o 4 bytes—. Finalmente en la línea 82 se pone en marcha el planificador.

En ocasiones las tareas se han de informar entre sí cuando ocurre cierto evento o cuando una tarea ha ocupado cierto recurso. Esta información es conocida como sincronización entre tareas. *FreeRTOS* dispone principalmente de dos tipos de elementos de sincronización conocidos como semáforos. Estos son los semáforos binarios y los de exclusión mutua —mutex—.

Los semáforos binarios son utilizados para que una tarea o interrupción informe a otra tarea que ha ocurrido un determinado evento y por tanto, esta última puede pasar de estar bloqueada a estar preparada para la ejecución. En este sentido podemos ver al semáforo binario como una bandera que indica que cierto suceso ha acontecido. Para el caso de interrupciones, es especialmente útil ya que en un sistema en tiempo real es necesario que las interrupciones no tomen mucho tiempo en ejecutarse. Por ello, una vez disparada una interrupción, la subrutina de atención a la misma ejecutará solo aquellas acciones imprescindibles —como informar al procesador que se está atendiendo la interrupción— para a continuación delegar en una tarea que verdaderamente atienda la interrupción.

El semáforo puede ser visto como una cola con un solo elemento de tamaño nulo: el testigo. De hecho, *FreeRTOS* implementa los semáforos a partir de colas. En el caso del semáforo binario, utilizado como «*bandera*», existirá una tarea o interrupción que generará y dará el testigo, informando que a partir de ese momento se puede llevar a cabo una tarea concreta. Por otro lado existirá la tarea que estará a la espera de recibir dicho testigo para realizar la función solicitada. Por ello, al igual que el ejemplo del código anterior, referente al manejo de colas, habrá un productor —tarea o interrupción— y una tarea consumidora. Para crear el semáforo binario se utiliza la función *xSemaphoreCreateBinary*, que no tiene argumentos de entrada y devuelve el manejador para el semáforo, que es de tipo *SemaphoreHandle_t*. El semáforo es creado vacío, es decir sin testigo, por tanto la tarea que lo espere estará inicialmente bloqueada. Para dar y generar un nuevo testigo están las funciones

xSemaphoreGive y *xSemaphoreGiveFromISR*. La primera genera el testigo en el semáforo desde una tarea y su único argumento es el manejador del semáforo al que se le quiere dar el testigo. La segunda es la versión segura para realizar lo mismo desde una subrutina de interrupción. Además del argumento de la función anterior, requiere un argumento opcional, de tipo «*portBASE_TYPE **», desde el cual la función pueda informa que el testigo que se ha dado, ha provocado que la tarea que se desbloquea —que es la delegación de la rutina de atención a la interrupción— tenga mayor prioridad que la tarea que se estaba ejecutando antes de entrar en la rutina de interrupción y por tanto se necesita que el siguiente cambio de contexto vaya desde la rutina de interrupción a la tarea en la que se delega, sin tener que pasar previamente por la tarea que inicialmente se estaba ejecutando. Este segundo argumento es un puntero que habrá que iniciar al valor «*pdFALSE*» y si se produce el desbloqueo de una tarea de prioridad mayor a la que estaba en ejecución, se devolverá el valor «*pdTRUE*», y habrá que utilizarlo en la llamada a la macro «*portEND_SWITCHING_ISR*» para poder hacer el cambio directo de contexto desde la rutina de atención a la interrupción a la tarea desbloqueada:

```
1  .
2  .
3  // Se declara globalmente el manejador del semáforo
4  SemaphoreHandle_t semaforoLCD;
5
6  // En alguna rutina de inicialización se crea el semáforo
7   semaforoLCD = xSemaphoreCreateBinary();
8  .
9  .
10 // Rutina de atención a la interrupción
11 // de la señal de sincronismo vertical del LCD
12 __irq void LTDC_IRQHandler( void )
13 {
14   if (LTDC->ISR & 1)
15   {
16     LTDC->ICR = 1;
17
18     portBASE_TYPE px = pdFALSE;
19     xSemaphoreGiveFromISR(semaforoLCD, &px);
20     // Imprescindible para que el cambio de contexto
21     // salte desde la subrutina de interrupción
22     // a la tarea en la que se delega
23     portEND_SWITCHING_ISR(px);
24   }
25 }
26
27
28 void TareaDelegadaIsrLCD( void * pvParameters )
29 {
30   while(1)
31    {
32      // La tarea queda bloqueada hasta que se lo diga la rutina de interrupción.
33      if( xSemaphoreTake(semaforoLCD, portMAX_DELAY) == pdTRUE )
34      {
35        // Se realiza la ejecución que correspondería a la rutina de interrupción.
36      }
37    }
38 }
```

Código 2.5: Cambio de contexto desde una ISR en FreeRTOS

En la línea 4 se declara el manipulador del semáforo a utilizar entre la subrutina de interrupción y la tarea. Esta declaración debe ser visible tanto para la interrupción como para la tarea. En la línea 7 se crea el semáforo binario. Está declaración estará dentro del cuerpo de una función utilizada para la inicialización. En la línea 19 se crea y entrega el testigo al semáforo desde la subrutina de atención a la interrupción. La llamada a esta función devolverá un valor «*pdTRUE*» en el puntero «*px*» —previamente ha de entregase a este puntero el valor «*pdFALSE*»—. Este valor devuelto se utilizará en la macro «*portEND_ SWITCHING_ISR*» para producir un cambio de contexto directo a la tarea «*TareaDelegadaIsrLCD*» —línea 28—. En la línea 33 la tarea delegada se quedará bloqueada hasta obtener el testigo que lanza la rutina de atención a la interrupción.

Los semáforos binarios se pueden utilizar también para la exclusión muta de tareas. En este sentido hay que crear un semáforo con un testigo y las tareas estarán diseñadas para bloquearse indefinidamente hasta tener acceso al testigo. Una vez que una de ellas tiene acceso al mismo, realizará su ejecución mantenido bloqueadas el resto de tareas que pelean por el testigo. Una vez que la tarea haya terminado su trabajo, debe devolver el testigo al semáforo para que otra pueda adquirirlo. Este tipo de patrón es habitual en el caso de acceso a un recurso compartido, en el que solo una tarea puede ocuparlo. El problema que presenta el semáforo binario es la carencia de inversión de prioridad, lo que puede provocar que una tarea de baja prioridad acceda a un recurso y momentos después una tarea de alta prioridad quede bloqueada por intentar acceder al mismo recurso. El problema surge cuando estando en esta situación, una tarea de prioridad media solicita al planificador su ejecución. En tal caso, ya que la tarea de prioridad media no va a acceder al recurso, que la tarea de prioridad baja tiene bloqueado, el planificador pasa a estado de ejecución la tarea de prioridad media bloqueando la tarea de prioridad baja. En consecuencia, se está ejecutando una tarea de prioridad media y está bloqueada una tarea de prioridad alta debido a que una tarea de prioridad baja tiene ocupado un recurso compartido. Para solventar esto está la inversión de prioridad, donde la tarea que ocupa el recurso adquiere momentáneamente la prioridad de la tarea más alta que quiere acceder a dicho recurso. Esto evita que la tarea de prioridad media se «cuele» en la ejecución. Para implementar esto *FreeRTOS* provee los semáforos «*mutex*» cuya diferencia con los binarios es esta habilidad comentada. No se entrará a discutir la creación y funcionamiento de estos semáforos ya que no aportan, desde el punto de vista del usuario, gran diferencia sobre los binarios y el tema cae fuera del alcance de este trabajo.

2.4 Modelo de memoria que FreeRTOS usa para TouchGFX

FreeRTOS tiene varios modelos de memoria para crear y destruir tareas u otros elementos. Todos ellos se basan en la declaración de una array en tiempo de compilación y en la implementación de las funciones de alojamiento y desalojamiento de dichas tareas u otros elementos dentro de este array, llamadas respectivamente «*pvPortMalloc*» y «*vPortFree*» y que *FreeRTOS* utiliza internamente. Estas dos funciones son llamadas dentro de otras funciones que, para el caso de tareas son: *xTaskCreate* y *vTaskDelete*, y que son manejadas por el usuario. El modo en que se implementen las funciones *pvPortMalloc* y *vPortFree* constituirán los diferentes modelos. En todos ellos, la dirección efectiva de inicio del array, que tratará *FreeRTOS*, será la inmediatamente posterior a la del inicio real que esté, alineada al tamaño del dato exigido para el alineamiento de la pila en el micro utilizado. En el caso de procesadores *ARM*, que es el tratado en este proyecto, este alineamiento es de 8 bytes [11]. Este alineamiento es necesario para hacer el sistema determinista, ya que la dirección de alojamiento de un mismo dato puede no ser siempre la misma en distintos momentos, lo que puede provocar que en unos casos el dato esté alineado y en otros. El no tener en cuenta el alineamiento, podría producir distintos tiempos de ejecución en la lectura/escritura de un dato del array. Como la dirección utilizada de inicio del array va a ser mayor a la asignada por el compilador, *FreeRTOS* ha de tener en cuenta este desplazamiento y ajustar el tamaño que maneja del array para que no surja el desbordamiento al intentar escribir un dato. Por ello, el tamaño real de memoria disponible para *FreeRTOS*, cuando se especifica el tamaño del array mediante la definición «*configTOTAL_HEAP_SIZE*», hecha en el archivo de configuración *FreeRTOSConfig.h* de *FreeRTOS*, será algo menor al tamaño real de este array. En la siguiente figura se muestra lo anteriormente explicado:

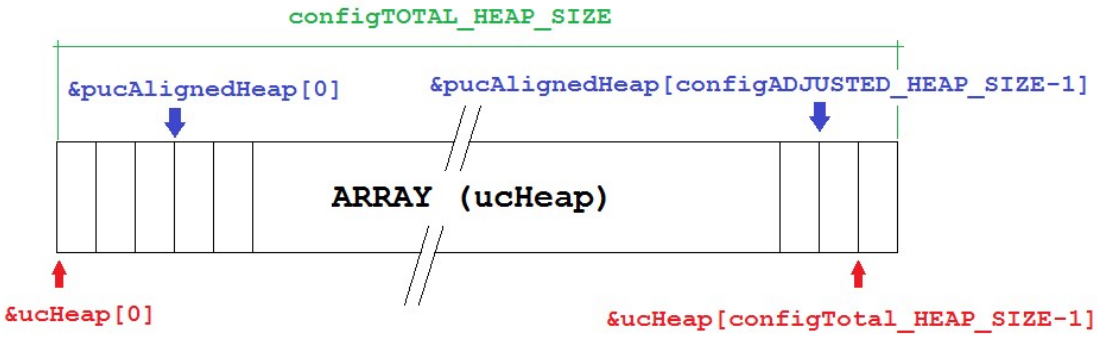


Ilustración 2.3: Gestión de la memoria por parte de FreeRTOS

En esta figura se muestra el array utilizado por *FreeRTOS* para alojar tareas y otros elementos cuyo nombre es «*ucHeap*». El tamaño de este, será el especificado en el «*define*» «*configTOTAL_HEAP_SIZE*». Cuando la dirección de inicio no está alineada, en este caso a 8 bytes, *FreeRTOS* toma como dirección de inicio aquella inmediatamente superior que si lo esté. Esta dirección queda almacenada en el puntero «*pucAlignedHeap*». Todos los elementos que se alojen en este array, deben ser de tamaño múltiplo del alineamiento, si no es así, el sistema reserva la cantidad de memoria solicitada más el relleno —«*padding*»— necesario para que lo sea. Esto es debido a que los datos alojados deben estar alineados a los 8 bytes —para no realizar más ciclos de los necesarios al leer/escribir un dato—, y con este relleno se asegura que la siguiente dirección a la que alojar un dato sí lo esté. Todo ello hace que no se pueda utilizar el tamaño total del array. El sistema corrige el tamaño del mismo en la definición «*configADJUSTED_HEAP_SIZE*», y lo tiene en cuenta para realizar las tareas de alojamiento y no rebosar el tamaño real del array.

Cada implementación de los modelos de memoria se guarda en un archivo diferente, así existen los archivos «*heap_1.c*», «*heap_2.c*», «*heap_3.c*», «*heap_4.c*» y «*heap_5.c*». En el primero de ellos, solo permite el alojamiento de tareas y otros elementos sin posibilidad de borrado de los mismos y por tanto, lo más habitual es usar este modelo para aplicaciones donde todas las tareas deban ser creadas al inicio de la ejecución. El segundo, permite alojar, borrar y realojar tareas, pero no compacta bloques libres, lo que puede provocar que se fragmente la memoria en pedazos pequeños si se producen muchos alojamientos/desalojamientos de memoria con tamaños aleatorios. Esto constituye un problema, ya que en el caso de darse tal situación, no se podrá alojar bloques de memoria más grandes de los bloques libres, aún cuando la suma total de memoria libre sea mayor a la del bloque a alojar. Este es el modelo que viene implementado por defecto en la distribución de *TouchGFX*. Será suficiente para la mayoría de las aplicaciones, ya que, normalmente solo se alojarán pocas tareas —la tarea del entrono gráfico y tal vez la tarea de la aplicación—. No se comentará el resto de modelos de *FreeRTOS* ya que no serán utilizados y su descripción, así como la descripción más profunda de los ya comentados, cae fuera del alcance de este proyecto.

2.5 Configuración del sistema operativo en TouchGFX

Para la configuración, *FreeRTOS* utiliza un archivo de cabecera llamado «*FreeRTOSConfig.h*». Este archivo no pertenece realmente al paquete del sistema, por ello, no está obligado a tener una localización en un directorio concreto, pero deberá especificarse su ruta dentro de la búsqueda de archivos de cabecera en el proyecto —en el caso de *Keil* 4.X o 5.X, esto se especifica en la ventana «Option for Target», accedida a través del menú «Project» y configurándolo en el cuadro de texto «Include Paths» dentro de la pestaña «C/C++»—. El nombre de esta cabecera sí ha de ser este, ya que se le hace referencia en el archivo principal de cabecera del sistema llamado «*FreeRTOS.h*».

La configuración de este archivo, lo divide en una serie de definiciones —«*define*»— que habilitan varias funciones del sistema o establecen el valor de cierto parámetro. Por ello nos encontramos con aquellas definiciones dedicadas a la configuración general de parámetros, que aparecen precedidas del prefijo «*Config*», aquellas dedicadas a incluir ciertas funciones, precedidas del prefijo «*INCLUDE_*», aquellas dedicadas al tratamiento de prioridades en interrupciones hardware, precedidas también del prefijo «*Config*», aquellas dedicadas al tratamiento de errores durante el desarrollo, aquellas dedicadas a realizar la correspondencia entre el nombre de interrupciones utilizadas en *FreeRTOS* y el utilizado por *CMSIS* , y aquellas utilizadas para el anclaje de funciones de retrollamada.

2.5.1 Directivas de configuración del archivo FreeRTOSConfig.h

Las definiciones para la configuración utilizadas en *TouchGFX* son las siguientes, de las cuales se comentarán las más importantes:

```
1  #define configUSE_PREEMPTION              1
2  #define configUSE_IDLE_HOOK               1
3  #define configUSE_TICK_HOOK               0
4  #define configCPU_CLOCK_HZ                ( SystemCoreClock )
5  #define configTICK_RATE_HZ                ( ( portTickType ) 1000 )
6  #define configMAX_PRIORITIES              ( ( unsigned portBASE_TYPE ) 5 )
7  #define configMINIMAL_STACK_SIZE          ( ( unsigned short ) 64 )
8  #define configTOTAL_HEAP_SIZE             ( ( size_t ) ( 5500 ) )
9  #define configMAX_TASK_NAME_LEN           ( 10 )
10 #define configUSE_TRACE_FACILITY          1
11 #define configUSE_16_BIT_TICKS            0
12 #define configIDLE_SHOULD_YIELD           1
13 #define configUSE_MUTEXES                 1
14 #define configQUEUE_REGISTRY_SIZE         8
15 #define configCHECK_FOR_STACK_OVERFLOW    2
16 #define configUSE_RECURSIVE_MUTEXES       1
17 #define configUSE_MALLOC_FAILED_HOOK      1
18 #ifdef SEMI_LOAD_MEASUREMENT
19 #define configUSE_APPLICATION_TASK_TAG    0
20 #else
21 #define configUSE_APPLICATION_TASK_TAG    1
22 #endif // SEMI_LOAD_MEASUREMENT
23 #define configTIMER_QUEUE_LENGTH          5
24 #define configTIMER_TASK_STACK_DEPTH      ( configMINIMAL_STACK_SIZE * 2 )
```

Código 2.6: Definiciones para la configuración de FreeRTOS

«*configUSE_PREEMPTION*» determina el modo de funcionamiento del sistema. Si es igual a «1», tendrá en cuenta las prioridades de las tareas, dando primacía a las tareas con prioridad preferente o repartiendo el tiempo de forma equitativa entre tareas de igual prioridad —*round-robin*—. Si es «0», las tareas funcionarán de forma cooperativa, es decir, que es la propia tarea la que cede la ejecución para que otras puedan ejecutarse.

«*configUSE_IDLE_HOOK*», si es «1», indica que se utilizará una función anclada a la tarea por defecto «*idle*», para que cada vez que no haya una tarea en ejecución, se ejecute y pueda medir —como lo hace *TouchGFX*— el tiempo en el que la CPU no tiene carga. La función a ser llamada ha de tener el siguiente prototipo: «*void vApplicationIdleHook(void)*». En «*configCPU_CLOCK_HZ*» se especifica la frecuencia de reloj en hercios, y como este dato ya se encuentra en la variable «*systemCoreClock*» declarada en «system_stm32f4xx.c» —archivo compilado internamente dentro de la librería *HAL* de *TouchGFX*—, se utiliza como valor de este «define». En «*configTICK_RATE_HZ*» se especifica el valor de cada «*tick*» o valor de tiempo en el que el sistema decidirá un cambio de contexto entre tareas expresado en hercios. El tipo de conversión explícita «(portTickType)» es un sinónimo para tipo «*long*» —32 bits—. «*configMAX_PRIORITIES*» determina el número de prioridades que manejará el sistema. Es conveniente especificar, exclusivamente, el número de prioridades a utilizar, ya que cada valor de prioridad consume RAM. «*configMINIMAL_STACK_SIZE*» establece el tamaño de la pila para la tarea por defecto —idle— especificado en palabras —de 32 bits—, es recomendable no bajar este tamaño. «*configTOTAL_HEAP_SIZE*» especifica la cantidad de RAM de la que dispone el núcleo de *FreeRTOS* para alojar las tareas y alojar posibles objetos creados dinámicamente por estas. El especificador de conversión explícita —*size_t*— es un sinónimo de «*unsigned int*». Esta definición ha sido ya mencionada en la sección «Modelo de memoria que *TouchGFX* usa para *FreeRTOS*», en este mismo capítulo. «*configMAX_TASK_NAME_LEN*» establece la longitud máxima de la cadena que pone un nombre descriptivo de la tarea. «*configUSE_16_BIT_TICKS*» determina, si es «1», que el contador de «*ticks*», desde el inicio de la ejecución que utilizará *FreeRTOS*, será de 16 bits. Si es «0», su tamaño será de 32 bits. Si «*configIDLE_SHOULD_YIELD*» es «1», significa que la tarea por defecto podrá compartir la misma rodaja de tiempo con otra tarea que tenga la misma prioridad, si es «0» la tarea por defecto esperará a que termine la tarea de igual prioridad y luego se ejecutará. «*configQUEUE_REGISTRY_SIZE*» establece el número máximo de colas y semáforos que pueden ser utilizados con fines de depuración. «*configUSE_APPLICATION_TASK_TAG*» habilita, si es «1», el uso de valores asignados a tareas o punteros a funciones para ser usadas como retrollamadas —callback—.

La configuración utilizada en un proyecto *TouchGFX* será el mostrado en el cuadro de código anterior con el ajuste, si ello es necesario, de la definición «*configTOTAL_HEAP_SIZE*».

2.5.2 Directivas de inclusión del archivo FreeRTOSConfig.h

Estas son las que comienzan con «INCLUDE_» y su objetivo es la incluir determinada funcionalidad:

```
1  /* Set the following definitions to 1 to include the API function, or zero
2  to exclude the API function. */
3  #define INCLUDE_vTaskPrioritySet          1
4  #define INCLUDE_uxTaskPriorityGet         1
5  #define INCLUDE_vTaskDelete               1
6  #define INCLUDE_vTaskSuspend              1
7  #define INCLUDE_vTaskDelayUntil           1
8  #define INCLUDE_vTaskDelay                1
```

Código 2.7: Directivas de inclusión en FreeRTOS

Así para este caso concreto, se habilita el uso de las funciones «*vTaskPrioritySet*», que permite modificar la prioridad de la tarea; «*uxTaskPriorityGet*», que permite obtener la prioridad de la tarea; «*vTaskDelete*», que permite borrar tareas; «*vTaskSuspend*», permite suspender la tarea, «*vTaskDelayUntil*», que permite esperar a cierto momento para que la tarea se despierte o «*vTaskDelay*», que mantiene la tarea durmiendo por un periodo de tiempo.

2.5.3 Directivas de manejo de prioridades en interrupciones en el archivo FreeRTOSConfig.h

Para el manejo de prioridades hardware, *FreeRTOS* utiliza dos definiciones. La primera de ellas es *configKERNEL_INTERRUPT_PRIORITY* que establece la prioridad de la interrupción del *tick* —en este caso concreto, la interrupción del Systick del cortex M4, encargada de los cambios de contexto—. Si solo se especifica esta definición en el archivo de configuración, toda interrupción que deba ser controlada por *FreeRTOS* ha de tener esta prioridad. Por otro lado está la definición *configMAX_SYSCALL_INTERRUPT_PRIORITY*, que especifica la mayor prioridad que puede tener una interrupción para que pueda ser manejada por el sistema operativo. Si se especifica esta última definición, con la definición *configKERNEL_INTERRUPT_PRIORITY* se debe indicar el nivel de prioridad mínima —que corresponderá a la interrupción *tick*—. De este modo el nivel de prioridad hardware a ser controlado por el sistema quedará determinado por la definición *configMAX_SYSCALL_INTERRUPT_PRIORITY*.

Se entiende por «interrupción controlada por *FreeRTOS*», aquella que no va a interrumpir un cambio de contexto y desde la cual se puede llamar a funciones del API de *FreeRTOS* —funciones especiales del *API* terminadas con el sufijo «*FromISR*»—. Cualquier otra interrupción con mayor prioridad a la especificada en esta definición, no será controlada por el sistema, lo que significa que no se podrá llamar a ninguna función del *API* de *FreeRTOS* y podrá interrumpir el normal funcionamiento del mismo, lo que implica que pueda provocar el incumplimiento del tiempo real del sistema. Para aclarar todo esto se muestra la siguiente figura [10]:

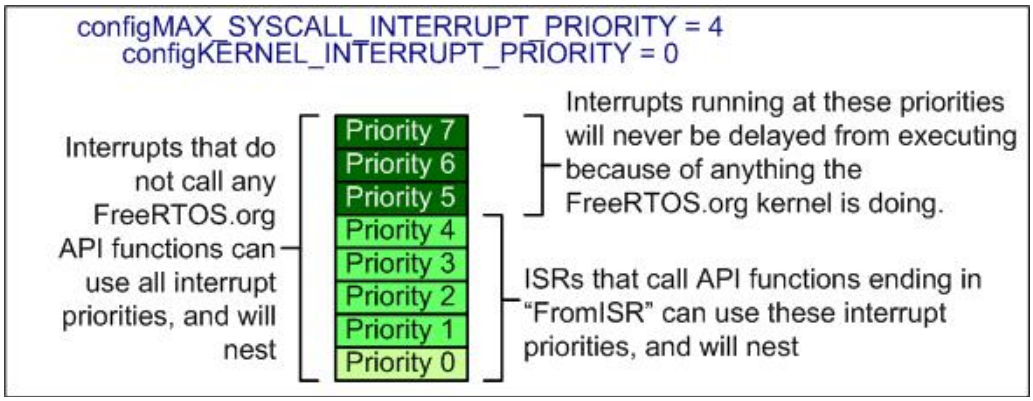


Ilustración 2.4: Esquema de interrupciones controladas por FreeRTOS

En esta figura se aprecia que los valores para *configKERNEL_INTERRUPT_PRIORITY* y *configMAX_SYSCALL_INTERRUPT_PRIORITY* son respectivamente 0 y 4, lo que significa que la prioridad para la interrupción *tick* es de cero, que es la más baja, y la máxima que puede gestionar el sistema es hasta interrupciones de nivel 4. Como se mencionó anteriormente, esto supone que las interrupciones con prioridades del «0» al «4» pueden ser enmascaradas por *FreeRTOS*, es decir, que su ejecución puede ser retardada para no interrumpir secciones críticas del sistema y además, estas interrupciones pueden utilizar funciones *API* especiales de *FreeRTOS* —aquellas terminadas en «*FromISR*»—. Por el contrario las interrupciones con prioridades 5, 6 y 7, no pueden ser enmascaradas por el sistema y no pueden utilizar funciones *API* del mismo. Estas últimas interrupciones se van a ejecutar sin retardo, interrumpiendo cualquier tarea —tenga la prioridad que tenga— o interrumpiendo cambios de contexto o secciones críticas.

Llegado a este punto, es necesario aclarar ciertos términos. Anteriormente se ha utilizado la palabra «prioridad» como sinónimo de «preferente», es decir, como aquello que tiene que ser tratado con primacía. Esto

no conllevaría ningún malentendido si a mayor valor numérico de prioridad le correspondiera mayor primacía en la ejecución de las interrupciones. Desafortunadamente esto no es así para los procesadores *ARM* con cortex M4. En estos, una interrupción con nivel de prioridad 5 tiene mayor preferencia que una interrupción con nivel de prioridad 7, sin embargo esta última tiene un valor numérico de prioridad mayor al primero. De hecho, el valor numérico de prioridad más preferente es el cero, valor de prioridad más bajo —preferencia más alta— que se le puede asignar a una interrupción. Por ello, a partir de ahora, se denotara con la palabra «preferencia» aquel nivel de interrupción que tenga primacía de ejecución sobre otra, mientras que se utilizará la palabra «prioridad» para denotar el baremo numérico de esta preferencia.

Lo comentado en el párrafo anterior constituye un problema para *FreeRTOS*, ya en el manejo de las tareas, son más preferentes aquellas que tienen valor numérico de prioridad más alto.

Para los procesadores *ARM* cortex M4, se especifica el valor de prioridad mediante 8 bits, de los cuales solo se implementan unos cuantos de la parte alta, mientras que los restantes de la parte baja se mantienen a cero y se ignora la escritura sobre ellos [9]. En concreto, para el procesador STM32F429ZI —cortex M4—, solo se implementa los cuatro bits más altos para especificar la prioridad. Dentro de la librería estándar de periféricos de STM32F4, está especificada la constante «__NVIC_PRIO_BITS» —declarada en el archivo «stm32f4xx.h», perteneciente a la librería citada y a *CMSIS*— cuyo valor es de 4, significando que los cuatro bits más altos de los ocho que especifican la prioridad, son los realmente usados. Esto hace que la prioridad menos preferente —valor numérico más alto de prioridad— sea 0xF0 en hexadecimal o de 240 en decimal, y la prioridad más preferente sea 0x00 —0 en decimal—, pero en pasos de 16, lo que hace un total de 16 niveles de prioridad. Para el manejo de esta circunstancia —valores de prioridad más preferente son los valores numéricos más bajos de prioridad—, *FreeRTOS* implementa otras tres definiciones más. A continuación se presenta la parte de código del archivo de configuración referente a lo tratado —que implementa la distribución de *TouchGFX*—, y se procede a comentar:

```
1  /* Cortex-M specific definitions. */
2  #ifdef __NVIC_PRIO_BITS
3  /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
4            #define configPRIO_BITS       __NVIC_PRIO_BITS
5  #else
6            #define configPRIO_BITS       6        /* 63 priority levels */
7  #endif
8
9  /* The lowest interrupt priority that can be used in a call to a "set priority"
10 function. */
11 #define configLIBRARY_LOWEST_INTERRUPT_PRIORITY            0x3f
12
13 /* The highest interrupt priority that can be used by any interrupt service
14 routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
15 INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
16 PRIORITY THAN THIS! (higher priorities are lower numeric values. */
17 #define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY       0x50
18
19 /* Interrupt priorities used by the kernel port layer itself.  These are generic
20 to all Cortex-M ports, and do not rely on any particular library functions. */
21 #define configKERNEL_INTERRUPT_PRIORITY (configLIBRARY_LOWEST_INTERRUPT_PRIORITY << (8-configPRIO_BITS))
22 /* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
23 See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
24 #define configMAX_SYSCALL_INTERRUPT_PRIORITY (configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8- configPRIO_BITS))
```

Código 2.8: Configuración de las prioridades tratadas por FreeRTOS

La primera de ellas es *configPRIO_BITS* —líneas 4 y 6—, que contendrá el número bits de la parte alta, de los ocho que maneja el micro concreto para la prioridad. En este caso no será igual a la definición *__NVIC_PRIO_BITS*, es decir, los 4 bits de la parte alta, ya que *FreeRTOS* no conoce esta constante. En este caso, *FreeRTOS* manejará 6 bits de la parte alta de la prioridad, ya que no está definida la constante anterior. La segunda definición adicional es *configLIBRARY_LOWEST_INTERRUPT_PRIORITY* —línea 11—, donde se especificará el valor de prioridad menos preferente —valor numérico mayor de prioridad—. Esta especificación de prioridad se hará teniendo en cuenta el valor numérico mayor que se puede representar con los bits que se implementan para la prioridad, sin tener en cuenta la posición que ocupen dentro de lo ocho bits que finalmente definen la prioridad. Así, para este caso, donde se contemplan 6 bits de prioridad —de forma errónea—, este

valor será el máximo posible a representar con 6 bits, esto es, 0x3f. El valor de esta constante es desplazado tantos bits a la izquierda como indica la operación 8 menos la constante *configPRIO_BITS* —en este caso, 8-6=2—, para especificar el valor real de la prioridad sobre los 8 bits. A continuación, es asignada a la constante *configKERNEL_INTERRUPT_PRIORITY* —línea 21—, que es la que realmente maneja *FreeRTOS* para la prioridad menos preferente —y que corresponde a la interrupción *tick* del sistema operativo—. Su valor, después del desplazamiento de dos bits, es de 0xfc. Sin embargo, el micro realmente solo tiene en cuenta los cuatro bits más altos, quedando realmente este valor a 0xf0.

Otra constante adicional implementada es *configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY* —línea 17—, encargada de contener el valor de prioridad más preferente —valor numérico menor de prioridad— a ser controlado por el sistema operativo —este valor es el representado sobre los seis bits en que, teóricamente, se maneja para la prioridad—. El valor que se especifica para esta constante, en el archivo *FreeRTOSConfig.h*, que suministra TouchGFX para la placa DISCO —con núcleo Cortex-M4— es de 0x50. Este valor es desplazado dos bits a la izquierda para especificar la prioridad sobre ocho bits, resultando el valor 0x40 y asignado a la constante *configMAX_SYSCALL_INTERRUPT_PRIORITY* que *FreeRTOS* maneja para la prioridad más preferente de una interrupción que pueda ser enmascarada —línea 24—. Este valor no se ve modificado al solo contemplar los cuatro bits más altos que realmente maneja el micro. Es decir, para la configuración tal y como está mostrada en la porción de código anterior, que es la que suministra TouchGFX para la placa DISCO, se tiene que la prioridad más preferente es el valor decimal 64, y la menos preferente 240. Teniendo en cuenta que realmente los cuatro bits bajos son siempre cero, hay un total de 12 prioridades diferentes que puede manejar *FreeRTOS*.

Sin embargo, esta configuración, aunque funciona, no es correcta, ya que está asumiendo que los bits que maneja el micro para la prioridad son seis, cuando realmente son cuatro. El manejo real de cuatro bits del micro para la prioridad, junto con los valores elegidos para las constantes, hacen que esta configuración funcione. En este proyecto se ha corregido esta configuración para reflejar los bits que realmente utiliza el micro —cuatro—. Para ello, se incluye en el archivo *FreeRTOSConfig.h*, la cabecera *stm32f4xx.h*, donde está definida la constante *__NVIC_PRIO_BITS*, y los valores para las constantes *configLIBRARY_LOWEST_INTERRUPT_PRIORITY* y *configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY* son modificados, respectivamente a 0x0f y 0x01, para que los valores de las constantes *configKERNEL_INTERRUPT_PRIORITY* y *configMAX_SYSCALL_INTERRUPT_PRIORITY*, sean respectivamente 0xf0 —240 decimal— y 0x10 —16 decimal—. Esto hace un total de 15 posibles niveles de prioridad para las interrupciones que maneja *FreeRTOS*. Se podría pensar que el valor de prioridad más preferente que pudiera utilizar *FreeRTOS* sería cero, valor de la constante *configMAX_SYSCALL_INTERRUPT_PRIORITY*. Sin embargo, hay que tener en cuenta FreeRTOS usa esta constante para enmascarar las interrupciones bajo su control. Para ello, FreeRTOS utiliza este valor como valor del registro *BASEPRI* del cortex M [10], y este registro enmascara todas las interrupciones con valor numérico de prioridad igual o mayor a su contenido. El valor cero de *BASEPRI* indica al cortex que no se va a enmascarar ninguna interrupción [9]. Por ello el valor de cero para la constante *configMAX_SYSCALL_INTERRUPT_PRIORITY* es un valor no permitido.

Una advertencia importante que hace la documentación de *FreeRTOS*, es la no utilización de subprioridades en el sistema de prioridades del cortex M. Tras el reset del micro, el sistema de prioridades queda configurado para no manejar subprioridades [9], pero hay ciertas situaciones en las que se puede dar el caso de tener configurado el micro con subprioridades. La más habitual, y que se advierte de ello en la documentación de *FreeRTOS* [10], se produce cuando se utiliza las funciones referentes al manejo del *NVIC* que implementa la librería estándar de periféricos del STM32F4 dentro del archivo «misc.c/h» —archivos pertenecientes a esta librería—. En este caso, la documentación de *FreeRTOS* recomienda, antes de ser utilizado, llamar a la función «*NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4)*», perteneciente a la librería estándar de periféricos. Otra posibilidad es hacerlo a través de la función «*NVIC_SetPriorityGrouping(0U)*» perteneciente a *CMSIS*.

2.5.4 Directivas de correspondencia entre los nombres de interrupciones y los utilizados por FreeRTOS

Son aquellas que emplea el sistema operativo para manejar internamente los nombres de las interrupciones que utiliza para su funcionamiento, y de esta forma independizar los nombres de las interrupciones respecto del micro concreto a utilizar. Las interrupciones utilizadas por *FreeRTOS* son: *SVC_Handler* que se dispara cada vez que se solicita el modo privilegiado del micro y que internamente «renombra» a *vPortSVCHandler*; *PendSV_Handler*, que se dispara cuando han acabado de ejecutarse todas las interrupciones pendientes y por tanto, el momento ideal para realizar un cambio de contexto después de la ejecución de interrupciones. Esta interrupción es «renombrada» como *xPortPendSVHandler*; finalmente la interrupción *SysTick_Handler*, «renombrada» a *xPortSysTick_Handler*, se utiliza por el sistema operativo como base de tiempos para establecer el momento donde se decide el cambio de contexto para el reparto de tiempos entre tareas.

```
1 /* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
2 standard names. */
3 #define vPortSVCHandler SVC_Handler
4 #define xPortPendSVHandler PendSV_Handler
5 #define xPortSysTickHandler SysTick_Handler
```

Código 2.9: Directivas de cambio de nombre de interrupciones usadas por FreeRTOS

2.5.5 Directivas de anclaje de funciones de retrollamada

Estas directivas del archivo *FreeRTOSConfig.h* utilizadas por *TouchGFX*, sirven para que este último pueda medir la ocupación de la *CPU*. Para que esto se lleve a cabo, es necesario conocer una serie de conceptos como son la forma en la que se informa a *TouchGFX* de los instantes en los que la *CPU* pasa a actividad o pasa a inactividad; cómo *FreeRTOS* ancla una función a la ejecución de la tarea por defecto —idle—; cómo *FreeRTOS* ancla una función, que puede recibir un argumento, a cualquier tarea y cómo *FreeRTOS* ejecuta estas funciones ancladas a tareas en los cambios de contexto.

El objeto *HAL* de *TouchGFX* —objeto que representa la capa de abstracción de hardware—, es el encargado, entre otras funciones, de calcular el tiempo de ocupación de la *CPU*. Para ello dispone de un puntero al objeto de la clase «*MCUInstrumentation*». Este último objeto es el encargado de habilitar en la *CPU* las utilidades de depuración, y en concreto, la encargada de habilitar la utilidad de cuenta de ciclos de reloj desde el inicio de la ejecución por parte de la *CPU*. Posee la función «*setMCUActive*» cuyo único argumento booleano será falso para indicar el momento de inicio de inactividad de la *CPU* o verdadero para indicar el inicio de actividad de la misma. Este método calcula los ciclos de ocupación del micro que pueden ser obtenidos a través de su otro método «*getCCConsumed*». También posee la función «*getElapsedUS*» que permite calcular el intervalo de tiempo en microsegundos transcurrido entre dos cantidades expresadas en ciclos, teniendo en cuenta el reloj del sistema expresado en MHz —este dato se obtiene de la variable *SystemCoreClock* dividida por 1e6, perteneciente al archivo «*system_stm32f4xx.c*» de *CMSIS*—. El objeto de la capa *HAL* implementa un método del mismo nombre —*setMCUActive*— con el mismo argumento, pero este método ha de tener en cuenta el tiempo en el que la *CPU* no está ocupada para poder hallar el tanto por ciento de ocupación de la misma. Para ello, utiliza una variable miembro, «*cc_begin*» y le pasa su argumento al método homónimo del objeto de la clase «*MCUInstrumentation*». En definitiva, para que *TouchGFX* pueda calcular el tanto por ciento de ocupación del micro, necesita que el sistema operativo, *FreeRTOS*, llame al método «*setMCUActive*» de la capa *HAL* de

TouchGFX, con argumento falso, en el momento en que entre en ejecución la tarea por defecto, y con argumento verdadero, en el momento en el que dicha tarea deja la ejecución:

```
1 //Llamar por FreeRTOS en el momento de abandonar la ejecución la tarea Idle
2 touchgfx::HAL::getInstance()->setMCUActive(true);
3
4 //Llamar por FreeRTOS en el momento de entrar en ejecución la tarea Idle
5 touchgfx::HAL::getInstance()->setMCUActive(false);
```

Código 2.10: Funciones de TouchGFX para medir la actividad del micro

Por otro lado, se necesita anclar la ejecución de una función a la tarea idle. Esto se hace mediante el siguiente código en el archivo *FreeRTOSConfig.h*:

```
1 #define configUSE IDLE HOOK 1
```

Código 2.11: Directiva de anclaje a la tarea idle en FreeRTOS

Esto informa a *FreeRTOS* que la tarea idle va a tener anclada la función cuyo prototipo ha de ser: *void vApplicationIdleHook(void)*. El problema de esta función es que no admite argumentos y no se puede informar a *TouchGFX* cuando entra y sale de ejecución la tarea idle.

FreeRTOS provee otro mecanismo adicional para anclar funciones a tareas, que sí admiten argumento. Estas funciones ancladas deben tener el siguiente prototipo: *portBASE_TYPE [nombre_función](void* p)*. Es decir, el argumento ha de ser un puntero de de tipo «*void*» y el tipo de retorno ha de ser *portBASE_TYPE* —para cortex M este tipo es un sinónimo de *long*—. Para llevar a cabo el proceso de anclaje y retrollamada, *FreeRTOS* dispone de las funciones *vTaskSetApplicationTaskTag* y *xTaskCallApplicationTaskHook*. La primera de ellas permite asignar un valor a una tarea —este valor se asigna a una variable interna que cada tarea posee cuyo nombre es «*Tag*»—. Admite dos argumentos, el primero, el manejador de la tarea a asignarle el valor —el manejador es una propiedad de cada tarea, que es determinada en la creación de cada una de ellas y que sirve para hacer referencia a las mismas después de su creación—, y el segundo el valor a ser asignado. De esta forma una tarea, llamando a esta función, puede asignar un valor a otra tarea. Si el primer parámetro es *NULL*, el valor se asigna a la propia tarea que llama a la función *vTaskSetApplicationTaskTag*. Este valor, además, puede ser un puntero a función —este puntero a función es de tipo *TaskHookFunction_t*, lo que significa que las funciones a ser llamadas han de tener el prototipo *portBASE_TYPE [nombre_función](void* p)*—. La segunda función *xTaskCallApplicationTaskHook*, permite ejecutar la función anclada a una tarea concreta. Para ello dispone de dos argumentos, el primero es el manejador de tarea cuya función anclada se quiere ejecutar, y el segundo es el valor a pasarle a la función anclada. Si el primer argumento es *NULL*, la tarea llama a la propia función anclada. Para poder utilizar este mecanismo de anclaje, es necesario especificar la siguiente definición en el archivo de configuración *FreeRTOSConfig.h*:

```
1 #define configUSE APPLICATION TASK TAG 1
```

Código 2.12: Directiva para anclaje a funciones en FreeRTOS

Para aclarar todo esto, se presenta el siguiente código genérico como ejemplo:

```
1  portBASE_TYPE funcionAncladaTarea1 ( void * pvParameter )
2  {
3     /* Código específico de la función anclada a la tarea 1 */
4
5     return 0;
6  }
7  ////////////////////////////////////////////////////////////////////////////
8  void Tarea1( void *pvParameters )
9  {
10    /* Se ancla a sí misma una función de retrollamada */
11    vTaskSetApplicationTaskTag( NULL, funcionAncladaTarea1 );
12
13    while(1)
14    {
15       /* Código específico de la Tarea1 */
16    }
17 }
18 ////////////////////////////////////////////////////////////////////////////
19 void Tarea2( void *pvParameters )
20 {
21    while(1)
22    {
23       if(condicion) //si se produce algún suceso que indique que se
24                     //ha de llamar a la función anclada a la Tarea1
25          {
26             xTaskCallApplicationTaskHook( manejadorTarea1, 0 )
27          }
28           /* Resto de código específico de la Tarea2 */
29    }
30 }
```

Código 2.13: Ejemplo de anclaje de función a tarea en FreeRTOS

En el cuadro anterior se muestra la función a ser anclada a la *Tarea1* —líneas 1 a 6— *funciónAncladaTarea1*, que tiene el prototipo esperado —retorno de tipo «*portBASE_TYPE*» y argumento de tipo *void**—. Dentro de la Tarea1 —líneas 8 a 17— se produce el «autoanclaje» de la función *funciónAncladaTarea1* mediante la llamada a *vTaskSetApplicationTaskTag*, ya que su primer argumento es *NULL*. Dentro del cuerpo de la *Tarea2* se llama a la función anclada a la *Tarea1* pasándole como argumento el valor cero. Para ello la *Tarea2* hace uso de la función de *xTaskCallApplicationTaskHook* especificando como primer argumento el manejador de la *Tarea1* —se supone que al crear la *Tarea1* se ha obtenido *manejadorTarea1* como su manejador— y como segundo argumento el valor cero que será el argumento recibido en la función anclada a la *Tarea1*, es decir, *funciónAncladaTarea1*.

Este mecanismo permite pasar argumentos a las funciones ancladas, pero se necesita de un mecanismo adicional para que sea el sistema operativo el que llama a la función anclada a la tarea idle en sus cambios de contexto. Para ello *FreeRTOS* dispone de una serie de macro funciones que el usuario puede redefinir y que son llamadas por el sistema en eventos característicos del mismo. Las dos que se van a tratar en este documento son *traceTASK_SWITCHED_OUT()* y *traceTASK_SWITCHED_IN()*. La primera de ellas, es llamada por *FreeRTOS* cuando una tarea abandona la ejecución, mientras que la segunda se llama cuando una tarea entra en ejecución. Para tener acceso a la tarea actual, *FreeRTOS* maneja la variable interna *pxCurrentTCB* consistente en un puntero a la tarea en ejecución. Así, cuando el sistema llama a la macro función *traceTASK_SWITCHED_OUT()*, el valor de *pxCurrentTCB* será el de la tarea que abandona la ejecución, mientras que cuando se llama a *traceTASK_SWITCHED_IN()*, el valor de *pxCurrentTCB* será el de la tarea que entra en ejecución.

Con todo lo expuesto, ya se puede anclar una función a la tarea idle; se puede anclar una función con un argumento y se puede llamar a esta función en los cambios de contexto. Ahora se ha de juntar todo para que se pueda alcanzar la funcionalidad buscada: informar al objeto *HAL* de *TouchGFX*, cuando la tarea idle entra y sale de la ejecución. Para ello en el archivo *FreeRTOSConfig.h* se especifican las siguientes líneas:

```
1  #define configUSE_IDLE_HOOK                      1
2  .
3  .
4  .
5  .
6  #ifdef SEMI_LOAD_MEASUREMENT
7  #define configUSE_APPLICATION_TASK_TAG           0
8  #else
9  #define configUSE_APPLICATION_TASK_TAG           1
10 #endif // SEMI_LOAD_MEASUREMENT
11 .
12 .
13 .
14 .
15 #ifndef SEMI_LOAD_MEASUREMENT
16 // To measure mcu load by measure time used in the dummy idle task
17 #define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, (void*)1 )
18 #define traceTASK_SWITCHED_IN()  xTaskCallApplicationTaskHook( pxCurrentTCB, (void*)0 )
19 #endif // SEMI_LOAD_MEASUREMENT
```

Código 2.14: Macrofunciones de ejecución en cabio de contexto

Primero se realiza la definición necesaria —línea 1— para informar a *FreeRTOS* que se va a anclar a la tarea idle la función cuyo nombre y firma es *void vApplicationIdleHook(void)*, y que debe estar definida en algún lugar dentro del proyecto. Entre las líneas 6 a 10 se realiza, de forma condicional, la definición necesaria para poder utilizar el anclaje a función con argumento, utilizando las funciones *vTaskSetApplicationTaskTag* y *xTaskCallApplicationTaskHook*. Finalmente, entre las líneas 15 a 19, se redefinen las macrofunciones que *FreeRTOS* llama en los cambios de contexto. En los cuerpos de estas redefiniciones están las llamadas a las funciones ancladas a la tarea actual en ejecución, pasando como argumento «1», para indicar que la tarea actual está saliendo de la ejecución o «0» si acaba de entrar en ejecución. Este código llama a todas las funciones ancladas de todas las tareas que entran y salen de ejecución, pero solo se va anclar una función con argumento a la tarea idle, por lo que finalmente solo la función con argumento anclada a la tarea idle será la llamada.

Aún queda por definir la forma en cómo una función con argumento se ancla a la tarea idle, ya que con la línea 1 del código anterior se ancla una función sin argumentos. También hay que definir el código de la función con argumento anclada que se comunique con el objeto *HAL* de *TouchGFX*. Todo ello se hace dentro del archivo *OSWrappers.cpp* de *TouchGFX*:

```
1  // FreeRTOS specific handlers
2  extern "C"
3  {
4   void vApplicationIdleHook( void )
5    {
6      vTaskSetApplicationTaskTag( NULL, IdleTaskHook );
7      while(1)
8      {
9      }
10   }
11 }
12
13 static portBASE_TYPE IdleTaskHook(void* p)
14 {
15   if ((int)p) //idle task sched out
16   {
17     touchgfx::HAL::getInstance()->setMCUActive(true);
18   }
19   else //idle task sched in
20   {
21     touchgfx::HAL::getInstance()->setMCUActive(false);
22   }
23   return pdTRUE;
24 }
```

Código 2.15: Llamada a las funciones Hal de TouchGFX desde FreeRTOS

Entre las líneas 1 a 11 está la definición de la función sin argumentos —*void vApplicationIdleHook(void)*— anclada a la tarea idle y que anteriormente se comentaba que debería estar definida en algún lugar. Para que se

ancle otra función con argumento a la tarea *idle*, se utiliza dentro de *void vApplicationIdleHook(void)* la llamada a la función *vTaskSetApplicationTaskTag* que auto ancla —debido a que el primer argumento es *NULL*— la función *IdleTaskHook*. Esta última función ya admite un argumento, y el valor recibido será el que le mande las redefiniciones de las macrofunciones *traceTASK_SWITCHED_OUT()* y *traceTASK_SWITCHED_IN()* que aparecen en el archivo *FreeRTOSConfig.h*. La función *IdleTaskHook* es la encargada de llamar al método *setMCUActive* del objeto *HAL* comunicándole cuando entra y sale de ejecución la tarea *Idle*, pasándole el valor «*true*», cuando la tarea *idle* abandona la ejecución y por tanto se inicia el tiempo de carga de la *CPU*, o «*false*», cuando la tarea idle entra en ejecución y por tanto se acaba el tiempo de carga de la *CPU*.

2.6 Implementación de la capa OSAL en TouchGFX

Ya que *TouchGFX* maneja directamente el funcionamiento del controlador del display —*LCD-TFT*—, las transferencias *DMA* gráficas —*DMA2D*— y la propia tarea del entorno gráfico, necesita un sistema de arbitración entre estos que garantice la sincronización. Para ello utiliza dos semáforos: uno para sincronizar el controlador *LCD-TFT* con la tarea del entorno gráfico, y otro para sincronizar esta última con el *DMA2D* —ver sección 1.5—. Estos semáforos no son accedidos directamente, sino que lo son a través de los métodos de una clase que hacen las veces de envoltura y que constituye la capa OSAL: la clase *OSWrappers*. Por supuesto, el sistema operativo utilizado es *FreeRTOS*, pero se puede utilizar otro cambiando el código de los métodos de la clase *OSWrappers*.

La definición de los manipuladores de los semáforos está realizada en el archivo *OSWrappers.hpp*, fuera del espacio de nombres «touchgfx», y fuera de la definición de la clase:

```
1  using namespace touchgfx;
2
3  static xSemaphoreHandle frame_buffer_sem;
4  static xQueueHandle vsync_q = 0;
5
6  // Just a dummy value to insert in the VSYNC queue.
7  static uint8_t dummy = 0x5a;
8
9  // Definición del cuerpo de los métodos de la clase
10 //  OSWrappers
11 . . .
```

Código 2.16: Definición de los semáforos de la capa OSAL

El semáforo *frame_buffer_sem* —línea 3— es de tipo binario, cuyo cometido es sincronizar la tarea gráfica con la ejecución del *DMA2D*. El hecho de no poderse ejecutar a la vez es debido a que ambos pueden acceder al buffer secundario de pantalla para actualizarle, y cuando uno de ellos accede a dicho buffer, debe evitar que el otro lo haga para no sobrescribir la información. Ambos toman este semáforo con el mismo método de *OSWrappers*, sin embargo, utilizan distintos métodos para devolverlo. Ello es debido a que en uno de los casos se devuelve desde una subrutina de atención la interrupción —*DMA2D*—, para asegurar el correcto cambio de contexto, mientras que en el otro, se devuelve desde la tarea gráfica.

El otro semáforo es *vsync_q* —línea 4—, implementado mediante una cola de un solo elemento, marca el inicio de la entrada en el pórtico trasero vertical por parte del periférico *LCD-TFT*, indicándolo a la tarea gráfica, pues es el momento de realizar el intercambio de búferes, si ello es posible. Esto lo hace enviando por la cola *vsync_q* el elemento, de tal forma que se desbloquea la tarea gráfica. Esta, al realizar su labor, se autobloquea, sacando el elemento de la cola y esperando a que la subrutina de interrupción envíe uno nuevo.

La definición de la clase *OSWrappers* es la siguiente:

```
1  #ifndef OSWRAPPERS_HPP
2  #define OSWRAPPERS_HPP
3
4  namespace touchgfx
5  {
6  class OSWrappers
7  {
8  public:
9      static void initialize();
10     static void signalVSync();
11     static void waitForVSync();
12     static void takeFrameBufferSemaphore();
13     static void tryTakeFrameBufferSemaphore();
14     static void giveFrameBufferSemaphore();
15     static void giveFrameBufferSemaphoreFromISR();
16 };
17 } // namespace touchgfx
18 #endif /* OSWRAPPERS_HPP */
```

Código 2.17: Definición de la clase OSWrappers

Por suerte en este caso, se dispone también del código fuente. El método *initialize* —línea 9— sirve para crear los dos semáforos y que sean accedidos mediante sus manipuladores. Para la sincronización entre el LCD-TFT y la tarea gráfica, se utilizan los métodos *signalVSync* —línea 10— y *waitForVSync* —línea 11—. El primero de ello es utilizado por la subrutina de atención a la interrupción del LCD para marcar el inicio de la representación —utiliza la función del sistema operativo *xQueueSendFromISR* para enviar el elemento por la cola, ya que se hace desde una ISR—. El segundo es utilizado por la tarea gráfica para coger el elemento de la cola —mediante la función *xQueueReceive* del RTOS— y autobloquearse a la espera del siguiente elemento —utilizando igualmente la función *xQueueReceive*—. Para la sincronización entre la tarea gráfica y el DMA2D, se utilizan los métodos *takeFrameBufferSemaphore* —línea 12—, *tryTakeFrameBufferSemaphore* —línea 13—, *giveFrameBufferSemaphore* —línea 14— y *giveFrameBufferSemaphoreFromISR* —línea 15—. El primero de ellos es utilizado tanto por la tarea gráfica como por la gestión del DMA2D para tomar el testigo del semáforo. Esto bloquea el acceso al buffer secundario de aquella entidad que no toma el testigo. El siguiente método es utilizado por la tarea gráfica para verificar si se puede coger el testigo —no bloquea—. Hay que recordar que la tarea gráfica además de gestionar las áreas a redibujar y los eventos pendientes de gestionar, también puede dedicarse a actualizar el buffer secundario —lo hace a través de los componentes que llaman a la función de la capa *HAL lockFrameBuffer* para asegurar el acceso exclusivo al buffer secundario entre ellos—. El siguiente método es utilizado por la tarea gráfica para devolver el testigo una vez se haya acabado de actualizar el buffer secundario por parte de los componentes gráficos. Finalmente, el último método, es utilizado por la subrutina de interrupción del DMA2D, que señaliza el final de transferencia de datos, para devolver el testigo del semáforo.

Esta es la implementación de la capa OSAL que *TouchGFX* trae por defecto, implementada con *FreeRTOS*. La utilización de esta capa es totalmente transparente al programador. Solo es necesario conocer su funcionamiento en el caso de cambiar de sistema operativo o tratar de utilizar el entorno gráfico sin sistema operativo —ver sección 2.7—.

La clase *OSWrappers* tiene todos sus métodos estáticos, con lo que no es necesaria una instancia de clase para poder utilizarlos.

2.7 Utilización de TouchGFX sin sistema operativo

La tarea gráfica está representada en el método *taskEntry* del objeto de la capa *HAL* —ver sección 4.5.10— que es una función que nunca retorna. No se tiene acceso a ella ya que no se dispone de código fuente en la versión de prueba de *TouchGFX*, así que se necesita sustituir este método por uno creado al efecto. El fabricante aconseja el siguiente:

```
1  HAL::getInstance()->enableLCDControllerInterrupt();
2  HAL::getInstance()->enableInterrupts();
3    while(1)
4    {
5        OSWrappers::waitForVSync();
6        HAL::getInstance()->backPorchExited();
7    }
```

Código 2.18: Tarea gráfica sin sistema operativo

En las dos primeras líneas se configura, respectivamente, el *LDC-TFT* para que interrumpa, inicialmente, al entrar en el área activa —*enableLCDControllerInterrupt*—, y habilitar en el *NVIC* las interrupciones del *LCD-TFT* y *DMA2D* — *enableInterrupts* —. Dentro del bucle se llama al método de la clase *OSWarppers waitForVSync*, que bloquea el bucle hasta que se devuelva el testigo que se emplea en el semáforo entre la tarea gráfica —este bucle— y el *LCD*. A continuación se llama al método de la capa *HAL backPorchExited*, que intercambia los buffers y llama al método *tick* de la capa *HAL* que ejecuta las tareas periódicas del entrono gráfico.

Sin embargo, es necesario modificar el código de los métodos de la clase *OSWrappers* para que en vez de manejar semáforos, maneje variables:

```
1  #include <touchgfx/hal/OSWrappers.hpp>
2  using namespace touchgfx;
3
4  static volatile uint32_t fb_sem;
5  static volatile uint32_t vsync_sem;
6
7  void OSWrappers::initialize()
8  {
9    fb_sem = 0;
10   vsync_sem = 0;
11 }
12
13 void OSWrappers::takeFrameBufferSemaphore()
14 {
15   while(fb_sem) HacerAlgo1();
16   fb_sem = 1;
17 }
18 void OSWrappers::giveFrameBufferSemaphore()
19 {
20   fb_sem = 0;
21 }
22
23 void OSWrappers::tryTakeFrameBufferSemaphore()
24 {
25   fb_sem = 1;
26 }
27
28 void OSWrappers::giveFrameBufferSemaphoreFromISR()
29 {
30   fb_sem = 0;
31 }
32
33 void OSWrappers::signalVSync()
34 {
35   vsync_sem = 1;
36 }
37
38 void OSWrappers::waitForVSync()
39 {
40   vsync_sem = 0;
41   while(!vsync_sem) HacerAlgo2());
42 }
```

Código 2.19: clase OSWrappers sin la utilización de sistema operativo

Los semáforos han sido sustituidos por las variables estáticas *fb_sem* —línea4—, para controlar el acceso al buffer secundario, por parte de la terea gráfica —el bucle while de la porción de código anterior— y el *DMA2D*, y *vsync_sem* —línea 5—, que hace las veces del semáforo para la sincronización entre el *LCD* y la tarea gráfica. Los métodos de la clase *OSWrappers* que devuelven el testigo del semáforo que arbitra el funcionamiento de la tarea gráfica y el *DMA2D*, ponen a cero la variable que representa ese semáforo, significando que el semáforo está libre —métodos *giveFrameBufferSemaphore* y *giveFrameBufferSemaphoreFromISR*—. Cuando toman el semáforo, ponen la variable a uno, significando que el semáforo está ocupado —métodos *takeFrameBufferSemaphore* y *tryTakeFrameBufferSemaphore*—. En el caso de la variable que actúa como semáforo entre el *LCD* y la tarea grafica, cuando se pone el testigo, la variable se pone a uno —recordar que este semáforo se realiza con una cola en el caso de utilizar sistema operativo, donde la interrupción del *LCD* pone elementos y la tarea gráfica los recoge—. Este es el caso del método *signalVSync*. En caso de recoger el testigo de este semáforo, se pone la variable a cero —método *waitForVSync*—. Cuando los métodos de *OSWrappers* tienen que bloquean la tarea gráfica, utilizarán, dentro del método correspondiente, un bucle while donde se ejecute de forma repetida una función. Así para el caso del método *takeFrameBufferSemaphore*, se utilizará la sentencia «while(fb_sem)HacerAlgo1();», donde la función «HacerAlgo1», es una función que realizará la actualización del buffer secundario por parte de aquellos componentes gráficos que no utilizan *DMA*. Para el caso del método *waitForVSync* utilizará la sentencia «while(!vsync_sem)HacerAlgo2();». El fabricante de TouchGFX recomienda que la función *HacerAlgo1* no dure más de 50 milisegundos y *HacerAlgo2* no dure más de 5 milisegundos.

En este proyecto se ha utilizado sistema operativo, no siendo necesario realizar ningún tipo de cambio en la capa *OSAL*. De hecho, el conocimiento de la utilización del entorno gráfico sin sistema operativo, tal y como se

muestra en esta sección, se produjo mucho después del inicio del mismo —como un año después, debido a que el fabricante de *TouchGFX* no especifica nada en el manual y la información sobre ello no apareció hasta entonces en su página web—.

3 Implementación física del proyecto

3.1 Introducción

Una vez vista la estructura del entorno gráfico *TouchGFX* y su interacción con el sistema operativo *FreeRTOS*, es necesario un soporte hardware que sustente, físicamente, a todo el desarrollo. La concreción de este hardware se basará, principalmente, en la determinación del microcontrolador a utilizar, pero también en los periféricos asociados —tanto internos al micro como externos a él—. La opción más rápida y fácil es utilizar una placa de desarrollo suministrada por el fabricante del micro, que cumpla tanto con los requisitos hardware como con el soporte completo facilitado por *TouchGFX*. En el otro extremo está la creación de una placa propia, con la consecuente dificultad de fabricación y comprobación, así como el desarrollo de la capa HAL de *TouchGFX* para esta nueva placa. Como última opción está el término medio entre las dos anteriores: utilizar una placa soportada por *TouchGFX* pero a la que se le practiquen modificaciones. Esta última es la más versátil, ya que realmente es difícil que una placa determinada se ajuste totalmente a los requerimientos —una placa con un display grande, como la *STM32F469I-Discovery* posee pocos pines para propósito general, mientras que otra con más pines, como la *STM32F429I-DISCO*, tiene un display pequeño —. Sin embargo, esta última opción requerirá algún desarrollo en la capa HAL, concretamente, en aquellos componentes de esta capa referentes al recurso hardware modificado. Siguiendo criterios que mantengan un compromiso entre prestaciones y precio, se opta por la utilización de la placa *STM32F429I-DISCO*, soportada directamente por *TouchGFX*.

3.2 Descripción de la placa de desarrollo

Esta placa seleccionada — *STM32F429I-DISCO*— contiene el micro *STM32F429ZIT6* del fabricante *ST*, que alberga gran cantidad de periféricos internos —varios de ellos usados en este proyecto, tanto directamente, como indirectamente a través de *TouchGFX* (usados de forma transparente)—. También contiene varios periféricos externos al micro, pero realmente, apenas se utiliza alguno de ellos. Sus periféricos externos más importantes, desde el punto de vista de este proyecto, son la memoria RAM externa, y el *display* táctil de carácter resistivo. Sin embargo, el tamaño de este último se considera insuficiente para la aplicación a desarrollar —un sistema que muestre la FFT de datos temporales capturados—, por tanto, será necesario realizar algunas modificaciones en la placa que traerán consigo el desarrollo software correspondiente.

3.2.1 Características de la placa

El aspecto de la placa STM32F429I-DISCO es el siguiente:

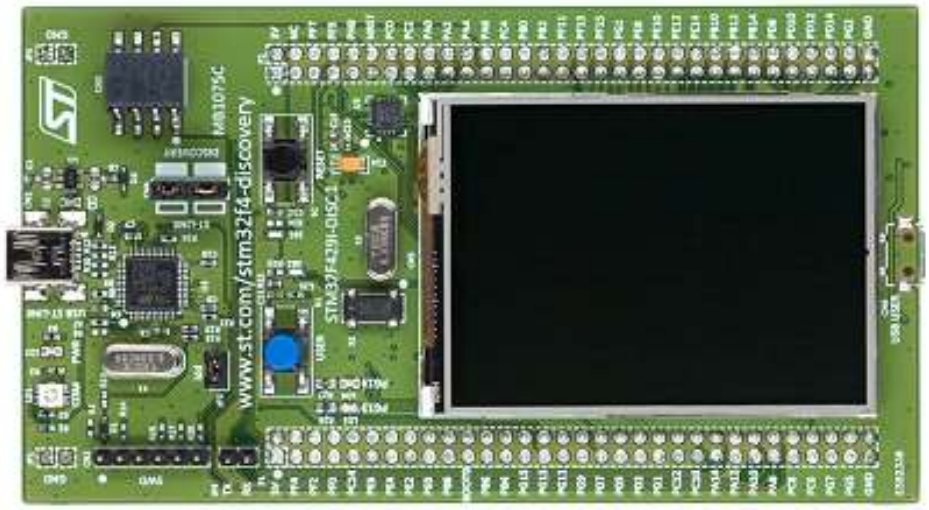


Ilustración 3.1: Apariencia de la placa STM32F429I-DISCO

En esta ilustración aparece la cara del display, por la otra se encuentra el micro y la memoria RAM externa. Su diagrama de bloques es el siguiente:

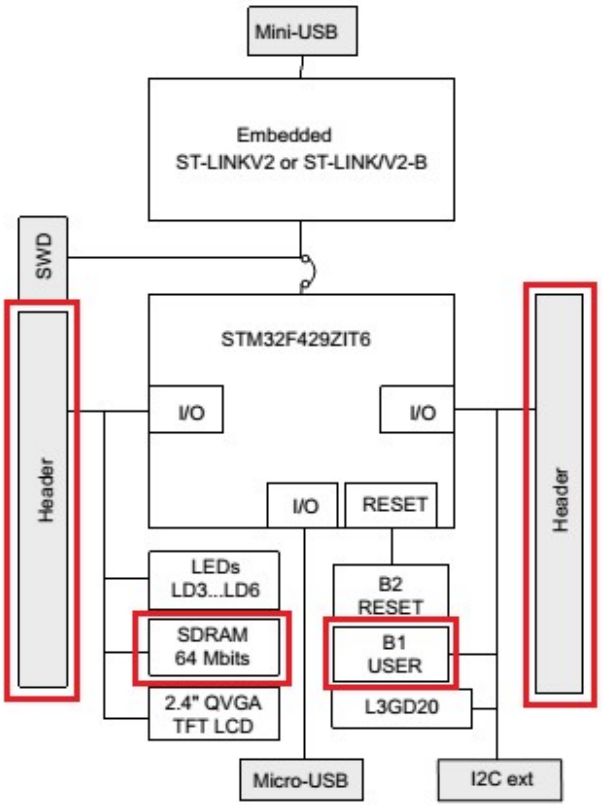


Ilustración 3.2: Diagrama de bloques de la placa STM32F429I-DISCO

Dispone de un puerto mini-USB —parte superior de la ilustración— por donde se puede alimentar a la placa y programarla; de un sistema de depuración integrado —ST-LINKV2—; del micro —*STM32F429ZIT6*—; de seis *led*, uno para indicar comunicación con el PC, otro para indicar alimentación, otro para indicar conexión de dispositivo en el conector Micro-USB —parte inferior de la ilustración—, otro para indicar desbordamiento en esa conexión y dos conectados a los puertos PG13 y PG14, para usos generales; de la memoria RAM de 8 Mbytes; del *display* de 2,4 pulgadas, del giroscopio L3GD20; de dos pulsadores, uno de usuario, de color azul y otro para el reset, de color negro, y de dos conectores —*Header*— de tipo tira de pines doble que conectan directamente con los pines del micro —situados a los laterales de la ilustración— . Ya que los periféricos externos —RAM, *display*, …— se conectan directamente al micro, sus señales también son accesibles por estos dos conectores, esta característica será utilizada a la hora de conectar un nuevo *display* de dimensiones mayores al ya instalado.

Los elementos de este esquema de bloques, que son utilizados para el proyecto, son los señalados con un marco en color rojo, es decir, la RAM externa, el pulsador de usuario, los conectores laterales donde están presentes las señales del micro, y por supuesto, el micro y el conector de programación y alimentación de la placa de tipo Mini-USB. A parte de estos periféricos externos, la placa dispone de otros que no están pensados para que el usuario acceda a ellos de forma directa, sino que son utilizados para controlar otro periférico externo a los que sí se tiene acceso. Este es el caso de integrado STMPE811, un integrado que sirve para expandir la entradas/salidas del micro, y que en la placa DISCO es utilizado para controlar la pantalla táctil del *display*. Este periférico será accedido directamente para poder controlar el sistema táctil del nuevo *display* que será instalado.

3.2.2 Características del micro

Las principales características del micro son las siguientes:

- Posee un núcleo Cortex –M4 con unidad de coma flotante.
- Frecuencia máxima de trabajo de 180 MHz, lo que permite 225 DMIPS.
- 2 MB de memoria Flash.
- 256 KB de memoria RAM interna —incluyendo 64 KB de memoria RAM acoplada directamente al núcleo—.
- Controlador de memoria flexible que permite conectar RAM externa de diferentes tipos así como memorias Compact Flash/NOR/NAND.
- Interfaz paralelo de tipo 8080/6800 para *displays* con controlador integrado.
- Periférico *LCD-TFT* para controlar *displays* sin controlador con una resolución máxima de *XGA*.
- *DMA2D*, acceso directo a memoria de forma bidimensional para transferencia gráfica.
- Capacidad para funcionar con osciladores externos en el rango de 4 a 26 MHz.
- Oscilador interno de 16 MHz para disciplinar todo el micro, ajustado en fabricación con una exactitud del 1%.
- Oscilador interno de 32 kHz para disciplinar todo el sistema, ajustado mediante red externo *RC*.
- Oscilador interno de 32 kHz dicado exclusivamente a controlar el periférico interno de tiempo real.
- Tres *ADCs* de 12 bits con una velocidad de muestreo de 2,4 MSPS en modo simple y de 7,2 MSPS en modo triple entrelazado.
- Dos convertidores DAC de 12 bits.
- Dos *DMAs* con un total de 16 flujos de transferencia.

3.2.3 Periféricos internos del micro utilizados

El esquema de bloques interno del micro STM32F429, resaltando aquellos periféricos internos utilizados, es el siguiente [4]:

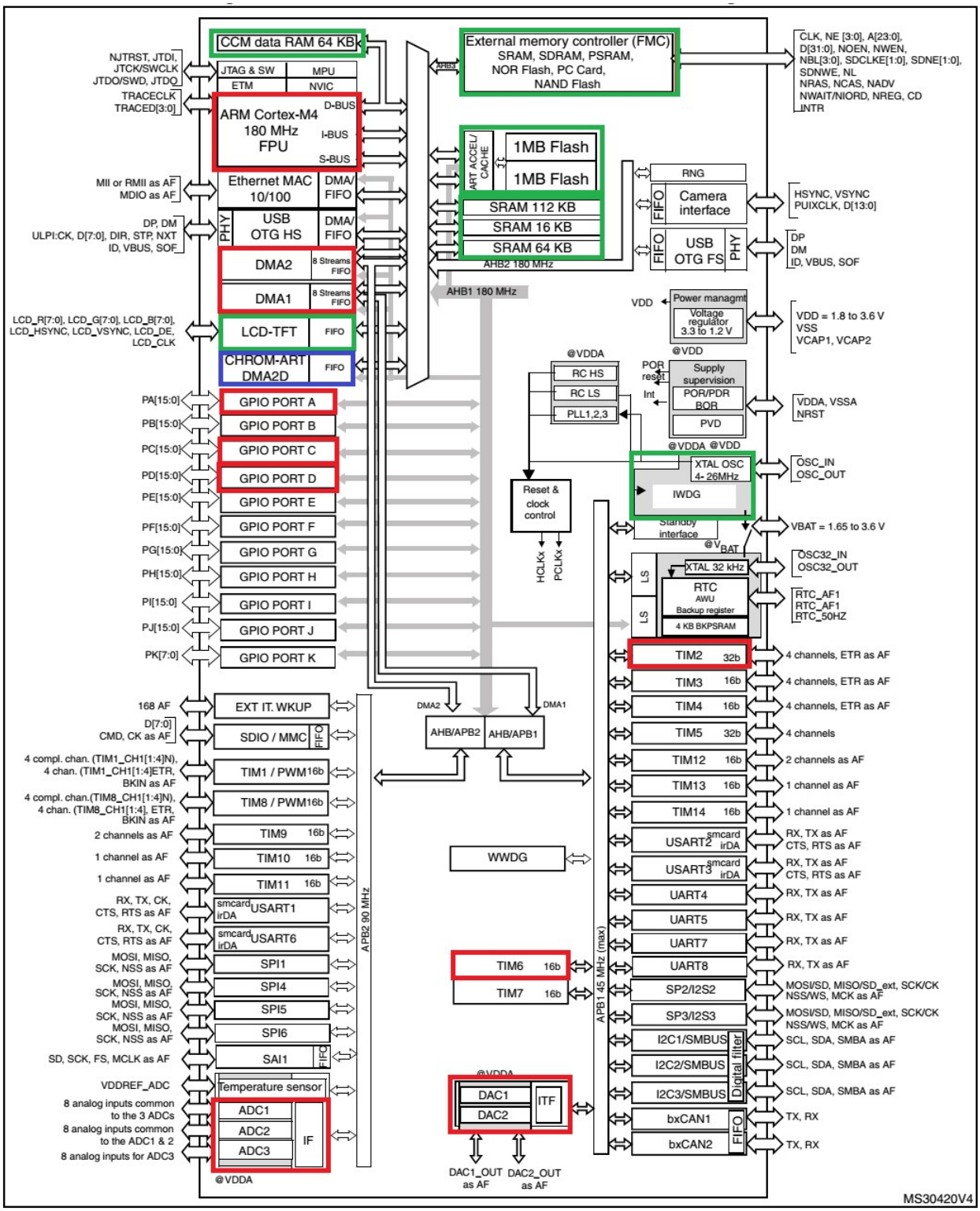


Ilustración 3.3: Esquema del micro STM32F429 con los periféricos internos utilizados

Los periféricos utilizados están remarcados con distintos colores. El color rojo indica la utilización directa de los periféricos, el verde, utilización indirecta a través de *TouchGFX* o utilización transparente, y el azul, utilización tanto directa como indirecta. Así los tres ADC son utilizados directamente por el objeto de la clase *DispositivoFFT* —ver sección 4.5.4—; los dos DAC son utilizados para el generador de señales configurado por la función *configuraGenerador* y manipulado mediante el botón de usuario a través de la rutina de atención a la interrupción *EXTI0_IRQHandler*, ambas contenidas en el archivo *inicializa_hard.c* —ver sección 4.3—; *TIM2* es utilizado para generar una señal de prueba cuadrada de 1 kHz configurada por la función *configuraPrueba* y la actualización de su salida —PD.12— se realiza a través de la rutina de atención a la interrupción *TIM2_IRQHandler*, ambas igualmente contenidas en el archivo *inicializa_hard.c*; *TIM6* es utilizado para disciplinar los dos *DACs* para realizar las peticiones *DMA*; DMA1 es utilizado en las transferencias de los *DACs*, mientras que DMA2 es utilizado en las trasferencias de los ADCs; GPIOA es utilizado para las salidas de los *DACs* —PA.4 y PA.5— y para el pulsador de usuario de la placa —PA.0—; GPIOC es utilizado para las entradas del conjunto de los *ADCs* —PC.2 PC.3— ; y GPIOD es utilizado como salida para *TIM2* —PD.12—. Por otro lado, el periférico DMA2D es utilizado en el normal funcionamiento de *TouchGFX* y directamente por el objeto de la clase *Graph* —ver sección 4.4.1.11— para dibujar las gamas de color entre la gráfica y la abscisa. Tanto la RAM interna como externa, es utiliza para albergar las variables y objetos utilizados. El periférico *LCD-TFT* es utiliza de forma trasparente a través de *TouchGFX*.

3.3 Descripción del display utilizado

Ya que el display que viene instalado en la placa STM32F429I-DISCO es muy pequeño —2,4 pulgadas—, se decide sustituirlo por otro mayor. Sin embargo hay que tener en mente el compromiso entre tamaño y posible cantidad de recursos consumidos —memoria RAM y ROM—. Además, debe tener capacidades táctiles de carácter resistivo, ya que se pretende reutilizar el hardware que la palca usa para el display que viene instalado en la misma. También se deberán revisar los tipos de señales que este nuevo display usa —sobre todo los de control—, así como características eléctricas en general, y en concreto las usadas para la retroiluminación. Una pantalla que cumple estos requisitos, es la mostrada en la siguiente figura [24]:

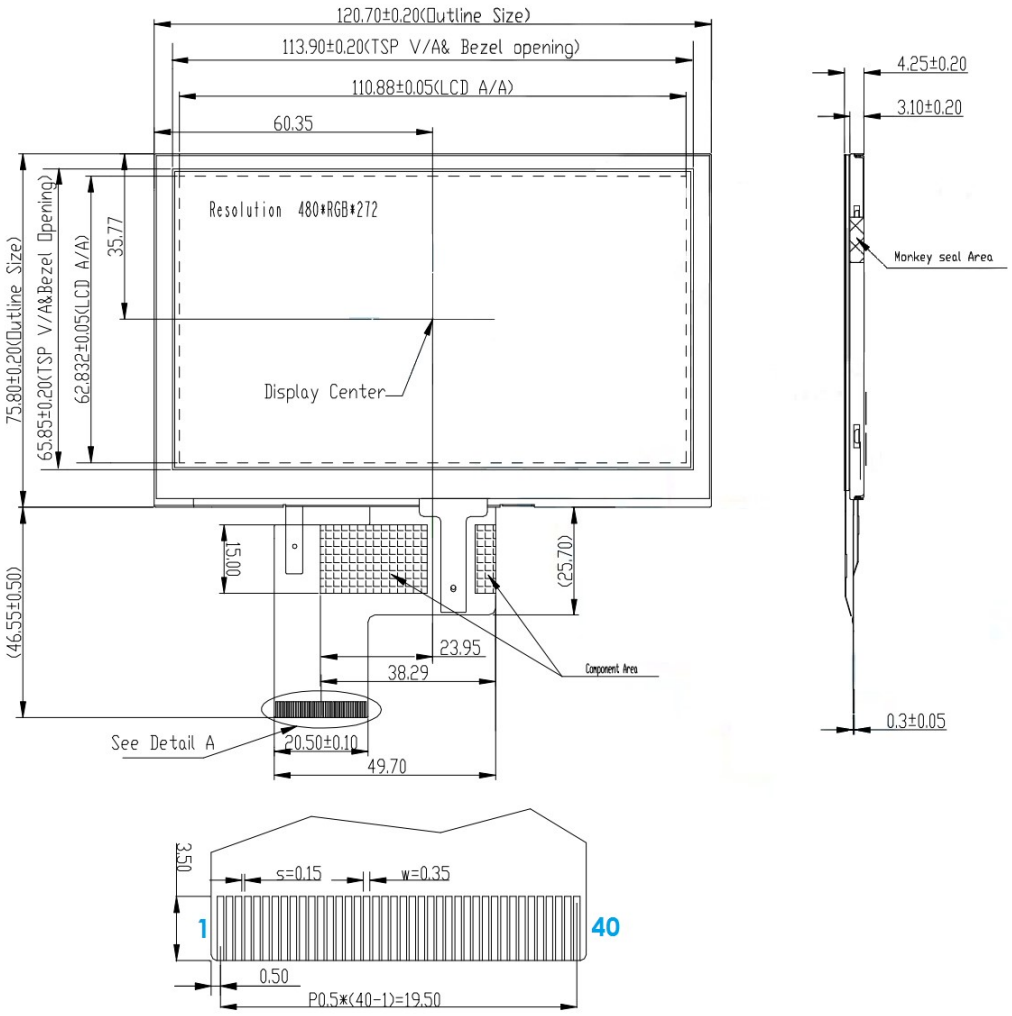


Ilustración 3.4: Display Innolux AT050TN33 utilizado en el proyecto

Se trata del display AT050TN33 del fabricante InnoLux, cuyas principales características son: es de un display de 5 pulgadas, tiene una resolución de de 480x272 pixeles, su profundidad de color es de 24 bits —ocho bits para cada componente de color—, dispone retroiluminación a leds y va equipado con una pantalla táctil de carácter resistivo. Tal y como se muestra en la figura, para acceder a las señales —de control, de datos, táctiles y retroiluminación—, dispone de una tira impresa de circuito flexible de 40 pines, donde la separación entre ellos es de 0,5 mm. Debido a que este tamaño de conector no es manejable, se va a usar un adaptador de circuito

impreso flexible —FPC— de paso 0,5 mm a conector DIP 40, cuyo paso es de 2,54 mm —o 0,1 pulgada—, que es el tamaño habitual en tecnología THT[15]. El aspecto de este adaptador, con la tira de pines hembra ya soldada es el siguiente:

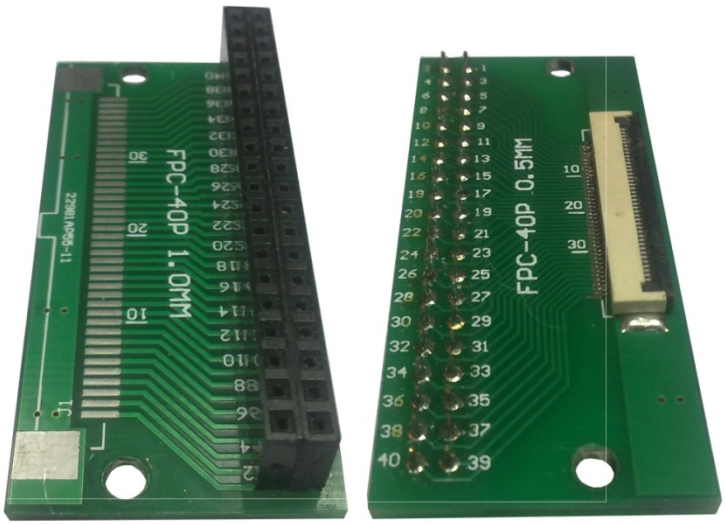


Ilustración 3.5: Adaptador de FPC 0,5mm a DIP40

En esta figura se muestran las dos caras del circuito impreso adaptador. Por un lado —la parte izquierda de la figura— está el conector DIP40, que será insertado en la placa que servirá de base para conectar tanto el display como la placa DISCO, y por otro —el lado derecho de la figura—, donde está el conector de tipo FPC para 0,5 mm. Este adaptador también permite utilizar un conector FPC de 1 mm —parte izquierda de la ilustración— pero no es usado en este proyecto.

El significado de la función de cada pin, en el cable plano flexible de este display es el siguiente:

Número de pin	Función	Número de pin	Función
1	Retroiluminación (cátodo)	31	Display on/off
2	Retroiluminación (ánodo)	32..33	No conectado
3	GND (alimentación)	34	DE (habilitación datos)
4	Vdd (alimentación)	35	No conectado
5…12	Dato R (5 LSB, 12 MSB)	36	GND (alimentación)
13…20	Dato G (13 LSB, 20 MSB)	37	X1
21…28	Dato B (21 LSB, 28 MSB)	38	Y1
29	GND (alimentación)	39	X2
30	CLK (reloj de pixeles)	40	Y2

Tabla 9: Asignación de pines del display AT050TN33 de InnoLux

[15] Through-Hole Technology, tecnología en la que la inserción de componentes en el circuito impreso se hace a través de perforaciones.

Consta, principalmente, de dos pines para la alimentación de la retroiluminación —pines 1 y 2--, dos de alimentación del display —pines 3 y 4—, 24 de datos del punto a representar —del pin 5 al 28—, cuatro para la pantalla táctil —del pin 37 al 40— y dos de control; el reloj de representación de puntos en pantalla —CLK, pin 30— y la señal de habilitación de datos —DE, pin 34—.

Este display, al contrario de la mayoría, no se controla con las señales habituales de sincronismo vertical —VSYNC— y horizontal —HSYNC—, sino que se controla únicamente con la señal de habilitación de datos —DE—, y por supuesto, como en el caso de displays controlados con señales de sincronismo, con la señal del reloj de pixeles —CLK, a veces referida como PCLK—. Esto permite ahorrarse un pin de la salida GPIO del micro —Las entradas/salidas GPIO llegan a convertirse en un bien escaso al utilizarse un display y SDRAM externa en la placa DISCO—.

Los cronogramas de funcionamiento, respecto a la señal DE, son los siguientes [24]:

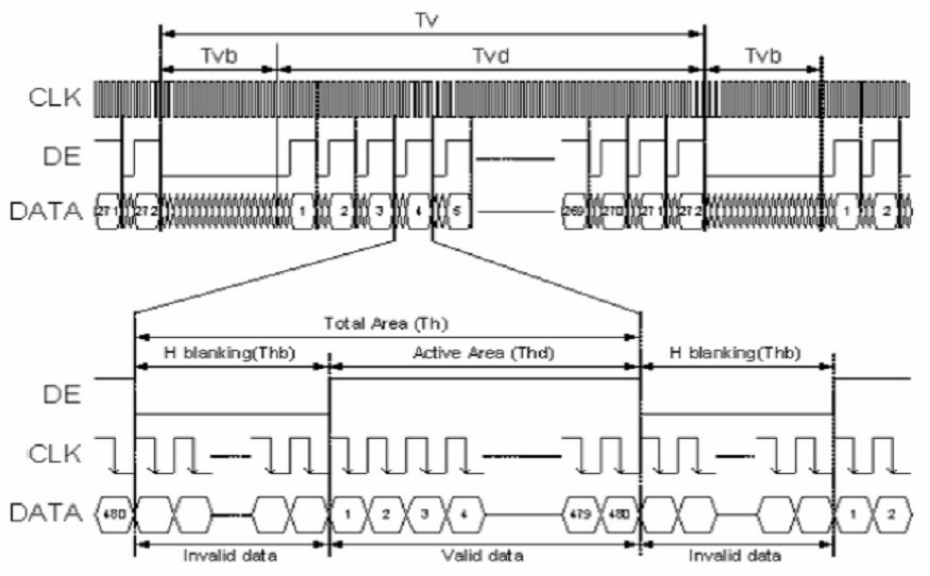


Ilustración 3.6 : Cronogramas de funcionamiento del display AT050TN33

En el cronograma superior se muestra la relación de la señal de habilitación de datos —DE— con la representación de las líneas de todo un cuadro en pantalla. Cuando la señal DE habilita los datos, cada uno de los puntos contenidos en cada línea —las líneas ejemplo se muestras numeradas como 1, 2, 3, 4, 5, …, 272— son pasados a pantalla a la velocidad que marca el reloj de pixeles —CLK—. El tiempo entre cuadro y cuadro viene fijado por el tiempo de borrado vertical Tvb, que informa a la pantalla que ha de prepararse para iniciar un nuevo cuadro. La duración de este cuadro está fijada por el tiempo denominado Tvd —alto del display en pixeles—. La suma de ambos tiempos constituye el periodo vertical, Tv.

En cronograma inferior se muestra la relación de la señal DE con los puntos de una línea —cada punto marcado como 1, 2, 3, 4,…, 480—. El tiempo de borrado horizontal está fijado por el parámetro Thb, que informa al display que ha de prepararse para iniciar una nueva línea. La duración de la línea, que corresponde con el ancho de pantalla, viene marcado por el tiempo Thd, y el periodo horizontal, por la suma de ambos, representado en Th.

Los parámetros de tiempo vertical se expresan en unidades de línea, mientras que los horizontales en puntos de pantalla —o periodos del reloj de pixeles—. Los valores típicos de este display, para estos tiempos y la frecuencia de envío de puntos a pantalla, son los siguientes:

Parámetro	Símbolo	Valor típico	Unidad
Periodo vertical	Tv	288	Líneas
Dimensión vertical	Tvd	272	Líneas
Borrado vertical	Tvb	16	Líneas
Periodo horizontal	Th	525	Puntos
Dimensión Horizontal	Thd	480	Puntos
Borrado Horizontal	Thb	45	Puntos
Frecuencia CLK	fclk	9	MHz

Tabla 10: Principales parámetros de temporización el display AT050TN33

Estos parámetros se han de tener en cuenta a la hora de configurar el periférico interno LTDC del micro STM32F429, que es el encargado de controlar el display. Este periférico provee varias señales de control, entre ellas DE, la interesante para este caso, sin embrago, la configuración se hace teniendo en cuenta el modelo de display controlado con señales de sincronismo vertical y horizontal. A la hora de la configuración, será necesario encontrar una conversión entre un modelo de display controlado con la señal DE y el controlado con las señales VSYNC y HSYNC. Esto será tratado en la sección de configuración del periférico LTDC —3.5.1.4—.

En cuanto a la retroiluminación led, el fabricante del display facilita la siguiente información (resumida):

Parámetro	Valor Típico
Tensión de retroiluminación led	19,8 V
Corriente de retroiluminación led	40 mA
Tiempo de vida (en el que la luminosidad decrece al 50%)	20000 h (mínimo)

Tabla 11: Características eléctricas de la retroiluminación del display AT050TN33

Aunque el fabricante no lo especifica, es muy probable que la retroiluminación esté constituida por dos ramas en paralelo de seis leds cada una, ya que cada led suele consumir unos 20 mA y la caída de tensión en cada diodo de este tipo suele ser de 3,3 V.

Para la parte táctil resistiva, se especifican las siguientes características:

Elemento	Valor
Resistencia del terminal X	De 100 a 900 Ω
Resistencia del terminal Y	De 100 a 900 Ω
Tensión de testeo	Típico 5V, Máximo 7 V

Tabla 12: Características eléctricas de la pantalla táctil del display AT050TN33

Estas características no son críticas, ya que se va a realizar una calibración de la pantalla táctil mediante un análisis de regresión que convierta directamente las cuentas de los ADCs, encargados de testear la pantalla, en posiciones de pixeles sobre la misma. La tensión mínima de comprobación no está especificada —entendiendo que no es crítico este valor para el buen funcionamiento de la pantalla—. Para el control de la pantalla táctil se reutilizará el integrado STMPE811 —un extensor de entradas/salidas y controlador de pantallas táctiles— que ya usa la tarjeta STM32F429I-DISCO para su display de 2,4 pulgadas. La tensión que este integrado usa para testear la pantalla táctil es a lo sumo de 3V ya que esa es su alimentación en la placa.

Para finalizar con las características del display, se resumen los valores máximos que no deben ser excedidos en ningún momento a riesgo que destruirlo. Estas son:

Elemento	Valor	
	Mínimo	Máximo
Tensión de alimentación	-0, 5V	5,0 V
Tensión en entradas	-0,5 V	5,0 V
Temperatura de operación	-10 ºC	60 ºC
Temperatura de almacenamiento	-20 ºC	70 ºC
Tensión inversa de LED	-	1,2 V (protegido con cener)
Corriente directa de LED	-	25 mA

Tabla 13: Valores máximos absolutos del display AT050TN33

Físicamente el display es alojado en una carcasa de metacrilato que además sirve de soporte para el adaptador FPC a DIP40. El aspecto del anverso y reverso del conjunto del display —donde se incluye la carcasa y el adaptador— es el siguiente:

Ilustración 3.7: Anverso del conjunto del display

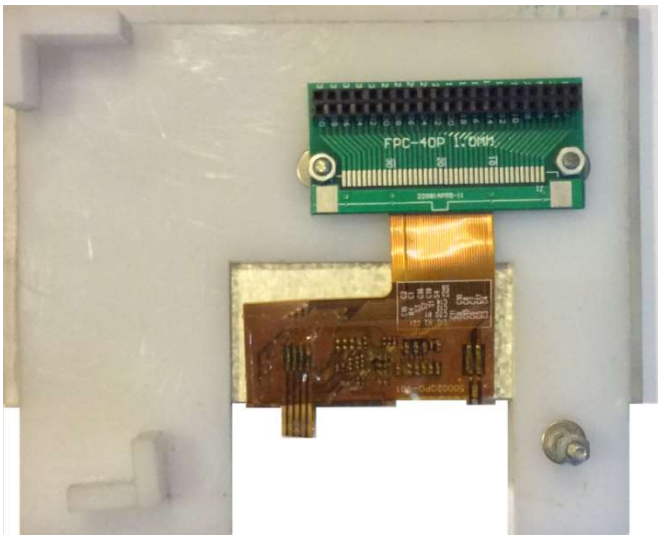


Ilustración 3.8: Reverso del conjunto del display

3.4 Instalación del nuevo display

3.4.1 Asignación de pines al display desde la placa 32F429IDiscovery.

Ya que se va a instalar un nuevo display a la placa 32F429IDiscovery —o también denominada STM32F429I-DISCO—, es necesario determinar qué pines de esta son lo que van a ir conectados a este display. Como la placa ya dispone de uno, y las señales de información y control son las mismas, independientemente del tamaño del display, se van a reutilizar [6]:

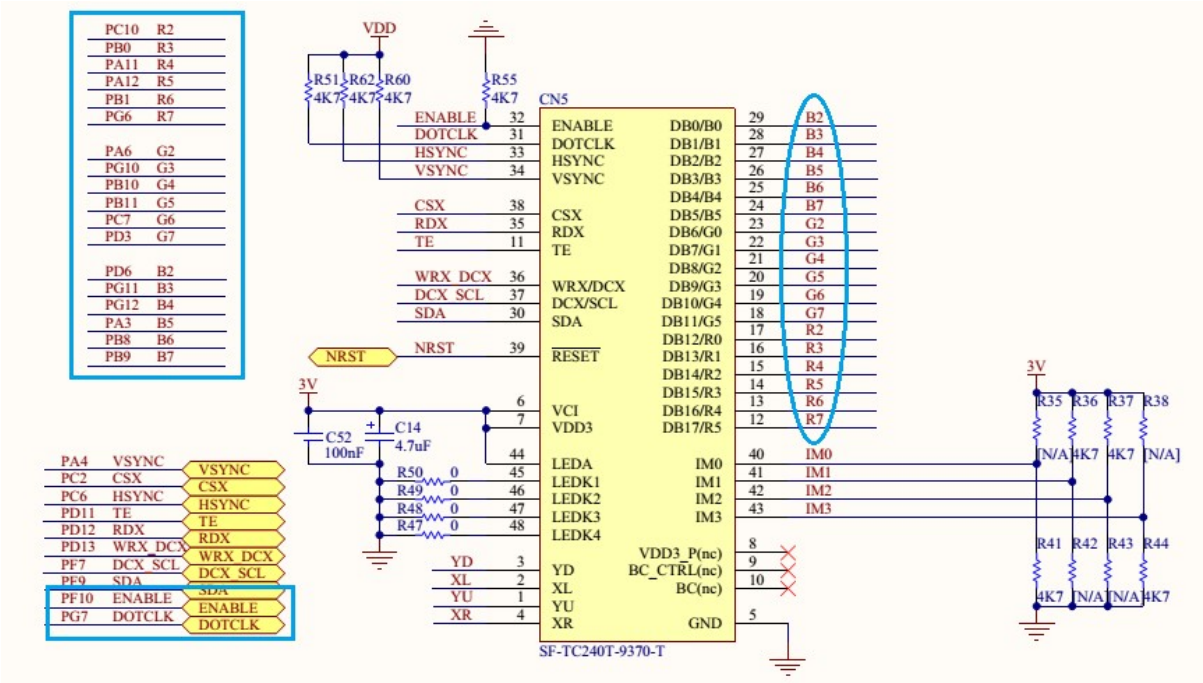


Ilustración 3.9: Asignación de pines en el display instalado en la placa DISCO

Esta ilustración ha sido construida a partir de los esquemas del manual de la placa STM32F429I-DISCO [6]. En ella se muestra el display y la conexión de sus señales a los pines correspondientes del micro. Las señales que van a ser de utilidad son aquellas enmarcas en azul, ya que el resto de señales, o son propias para el control del display instalado, o no son necesarias para el funcionamiento del nuevo display a instalar.

Estas señales también están presentes en los conectores P1 y P2 de la placa [6]:

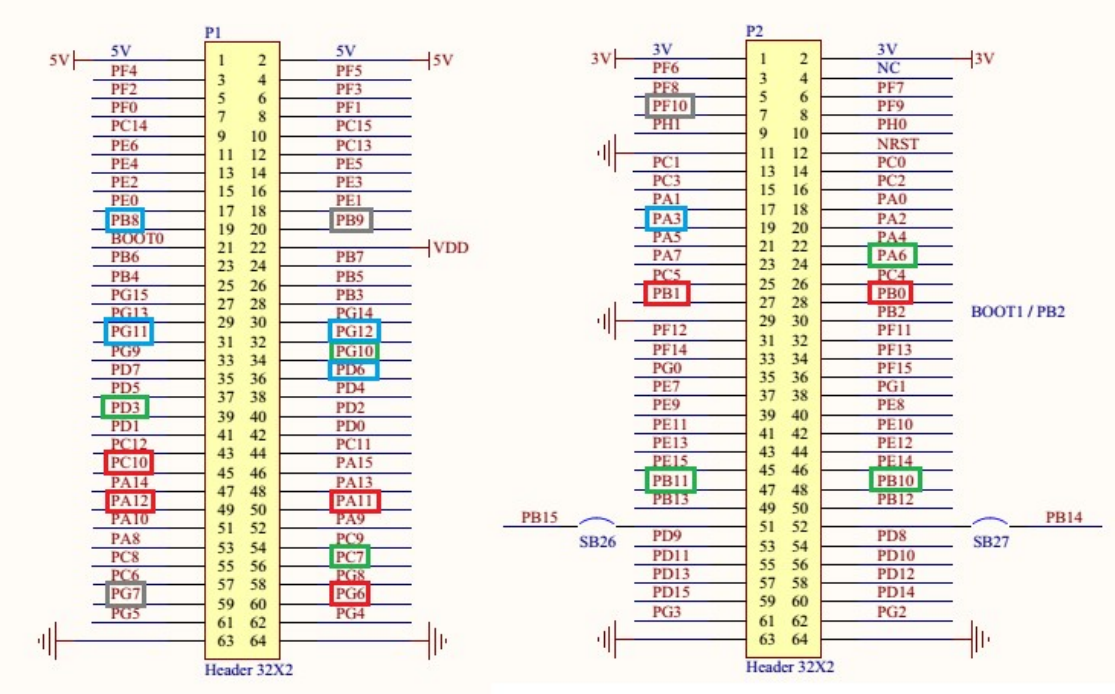


Ilustración 3.10: Señales para el nuevo display en los conectores P1 y P2 de la placa DISCO

Esta ilustración, al igual que la anterior, está realizada a partir de los esquemas de la palca DISCO. En ella se muestran enmarcadas las señales de datos con los colores correspondientes a cada componente, mientras que las señales de control se muestran en color gris. La información del bit de cada componente será mostrada en la siguiente sección —3.4.2— en que se muestra el esquema de la placa donde se conectará la placa DISCO junto con el nuevo display.

3.4.2 Esquema de la placa base.

Para poder conectar, de forma sólida, la placa STM32F429I-DISCO con el display AT050TN33, a parte de la carcasa de metacrilato que encierra a este último, se necesita una placa que conecte ambos. Se hace hincapié sobre que este soporte es el mínimo para poder realizar pruebas sobre la placa DISCO con el nuevo display instalado, y no debe ser tomado como diseño final —el propósito de este proyecto es la creación una aplicación con entorno gráfico y no el diseño del hardware—. Esta placa se denomina placa base, y además de unir los dos elementos citados, sirve para conectar el convertidor de la retroiluminación del display y para albergar la conexión de los pines de la pantalla táctil que irá conectada al ingenio, hecho con placas de de circuito impreso para prototipos, que se sitúa sobre el soporte del display original de la placa STM32F429I-DISCO —ver sección 3.5.1.6—. Por tanto, esta placa soporte consta de los terminales para conectar con la placa Disco, el terminal del nuevo display, el terminal de la pantalla táctil y los terminales del convertidor para la retroiluminación. El esquema del circuito es el siguiente:

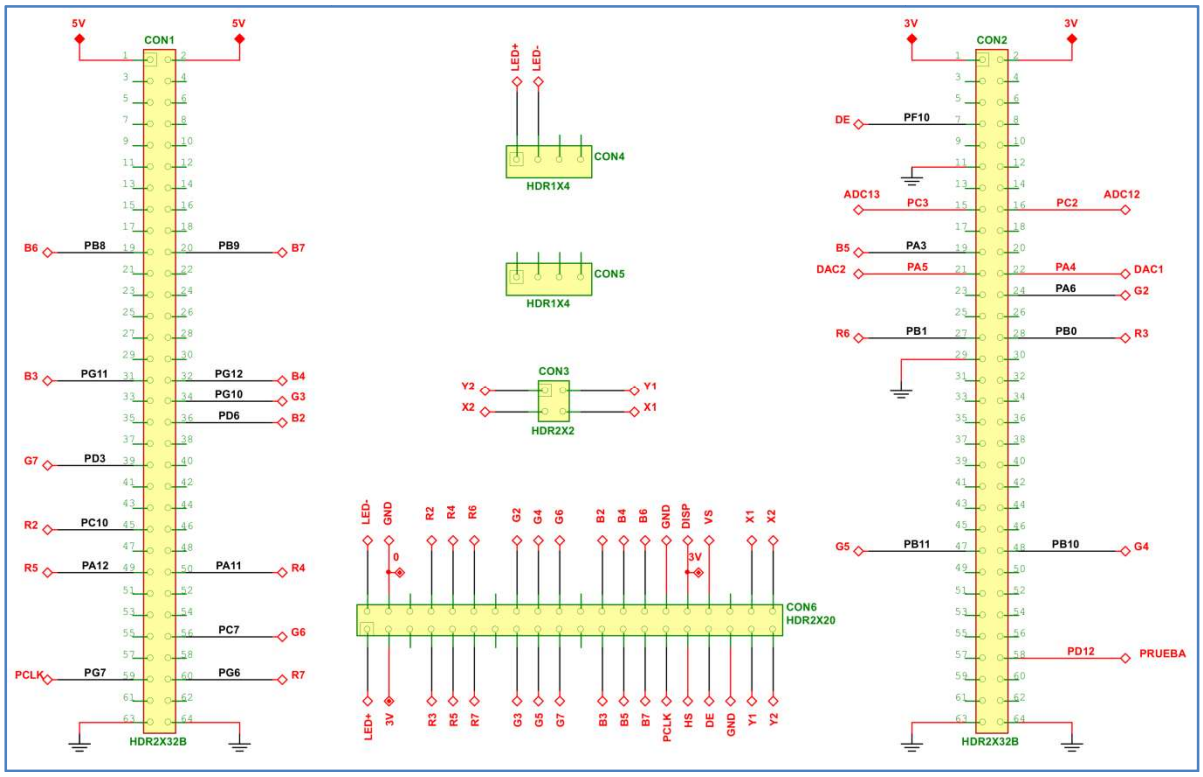


Ilustración 3.11: Esquema eléctrico de la placa base

Los conectores de 64 pines que aparecen en los laterales del circuito, CON1 y CON2, son los que se conectan a los terminales P1 y P2 de la placa DISCO, respectivamente. En ellos, solo se utilizan aquellos pines que tienen información referente al control y datos del display que ya son utilizados por la placa DISCO. Para facilidad de información, también aparecen los pines de señales que se conectarán directamente a la placa DISCO en la parte de sus terminales P1 y P2 que no están conectados a CON1 y CON2 de esta placa base —hay que recordar que los conectores P1 y P2 de la placa DISCO atraviesan esta, de tal forma que se tiene acceso a ellos desde ambas caras—. Estas señales son las referentes a las dos salidas del DAC —pines 21 y 22 de CON2—, las dos entradas del la configuración del ADC triple —pines 15 y 16 de CON2—y la señal cuadrada de prueba —en el pin 58 de CON2—. En CON3 se insertará el cable plano de cuatro hilos, cuyo extremo opuesto irá conectado al ingenio construido encima del alojamiento del display original de la palca DISCO —ver sección 3.5.1.6—. Los terminales CON4 y CON5 son los encargados de conectar el convertidor de retroiluminación. La salida de este está presente en CON4 — LED+ y LED- —, mientras que CON5 sirve exclusivamente de soporte y fijación del convertidor. La tensión de entrada al convertidor será suministrada a través de un conector que este dispone. Finalmente COM6 irá conectado al adaptador de FPC a DIP40 —ver sección 3.3— que está acoplado a la carcasa construida en torno al display instalado — AT050TN33— y que va conectado a su cable flexible de circuito impreso. Un posible diseño del circuito impreso para este esquema, realizado con la utilidad de autorrutado, y posterior retoque manual, del programa Ultiboard del fabricante NI, es el siguiente:

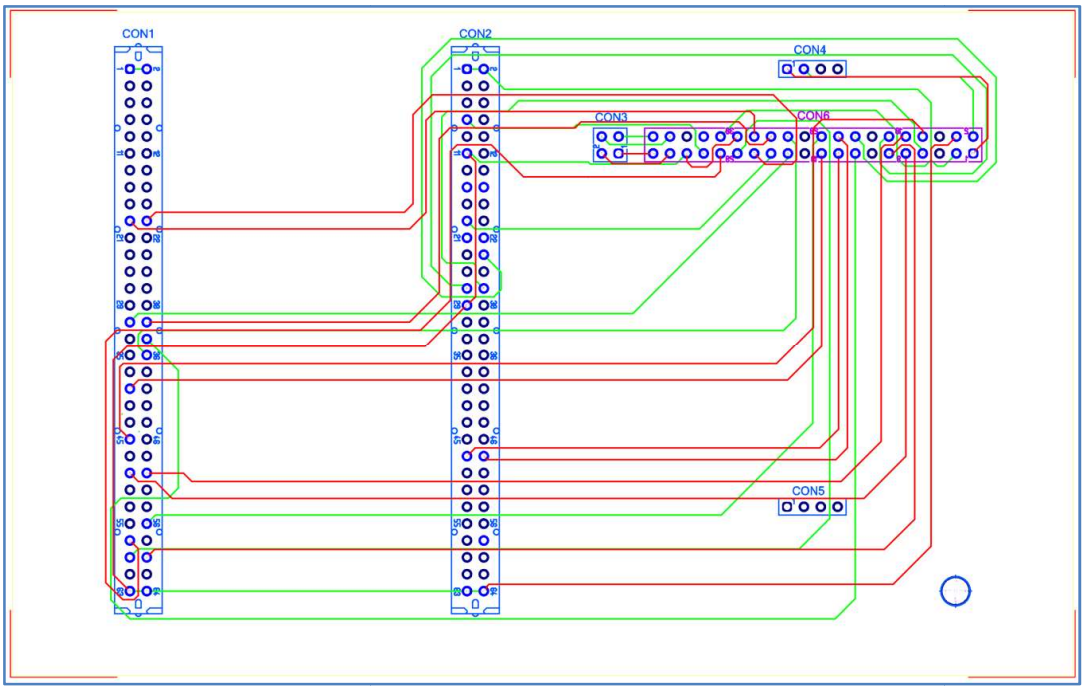


Ilustración 3.12: PCB de la placa base

El tamaño de la placa es Eurocard 100 mm x 160 mm y se trata de un diseño de doble cara. La vista de la placa está realizada desde la cara de cobre superior —color de pistas verdes—, donde van insertados los conectores CON1, CON2, CON3, CON4 y CON5. Mientras que en la cara inferior —color de pistas rojas—, va insertado el conector CON6. De esta manera, la placa DISCO, el convertidor y el cable plano de cuatro hilos de la pantalla táctil, van en un lado de la placa —en el esquema PCB mostrado, en la cara de arriba—, y el display va en la otra —la de abajo—. Solo CON1 y CON2 son conectores de tipo tira de pines hembra —para que pueda insertarse la placa DISCO—, mientras que el resto son de tipo macho.

Una visión en 3D que facilita el software utilizado —Ultiboard de NI— permite tener una idea del resultado final real:

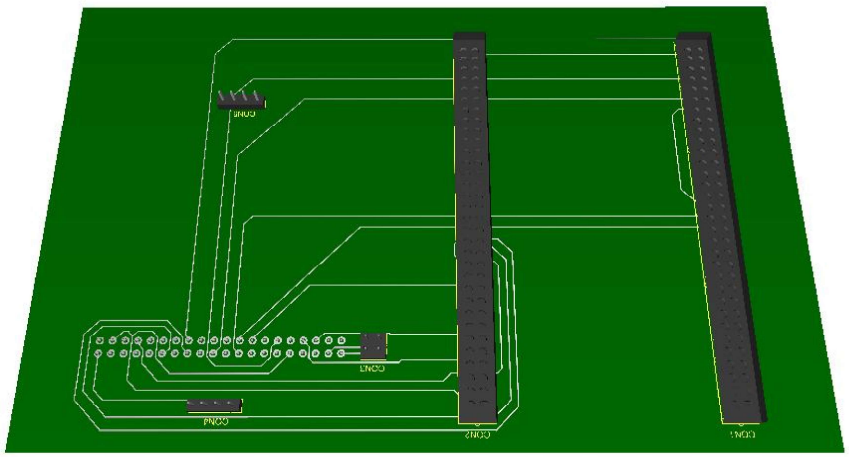


Ilustración 3.13: Visión 3D de la cara de arriba de la placa base

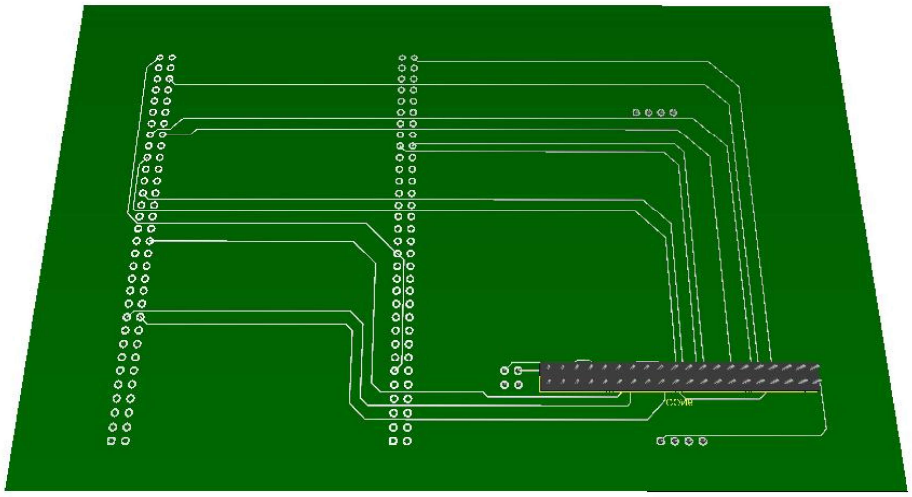


Ilustración 3.14: Visión 3D de la cara de abajo de la placa base

Como ya ha sido comentado, la finalidad del proyecto no es la realización de esta placa, y unido a criterios de practicidad, hace que se decida realizar el montaje sobre una placa protoboard de tipo perfboard con las mismas dimensiones Eurocard. Las conexiones serán realizadas tanto con cables como con caminos de soldadura, concretamente, en la cara superior serán utilizados caminos de soldadura, mientras que la inferior, cables. La apariencia de ambas caras es la siguiente:

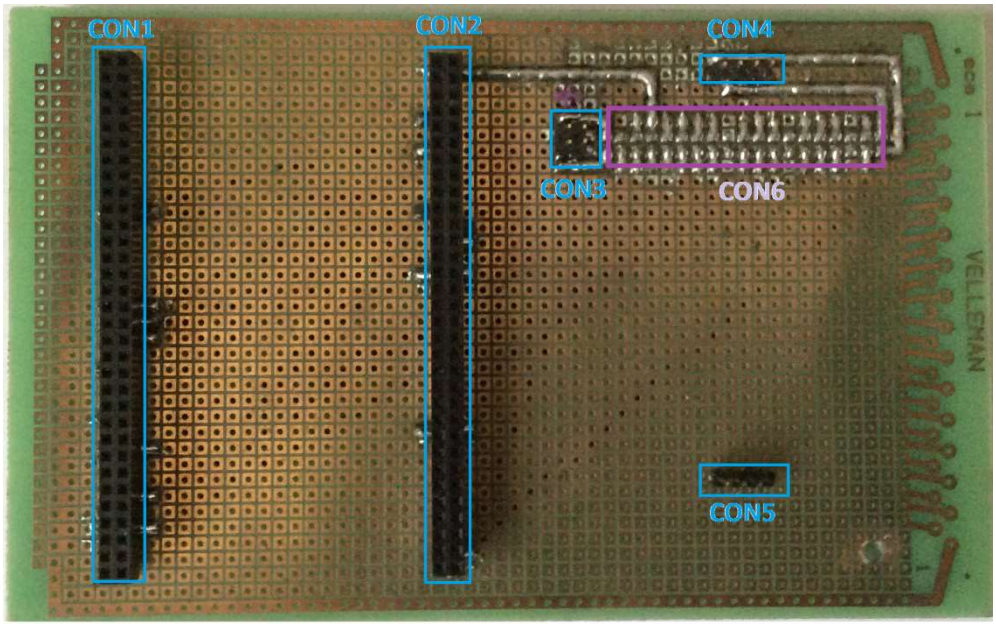


Ilustración 3.15: Cara superior de la placa base real

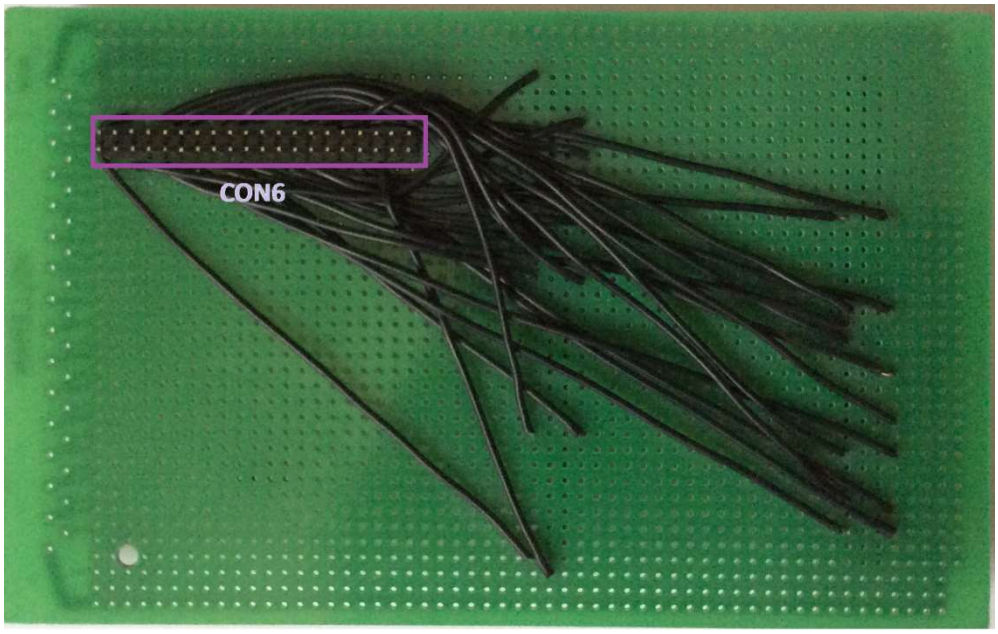


Ilustración 3.16: Cara inferior de la placa base real

El resultado de insertar todos los elementos en esta placa se muestra en las dos siguientes figuras:

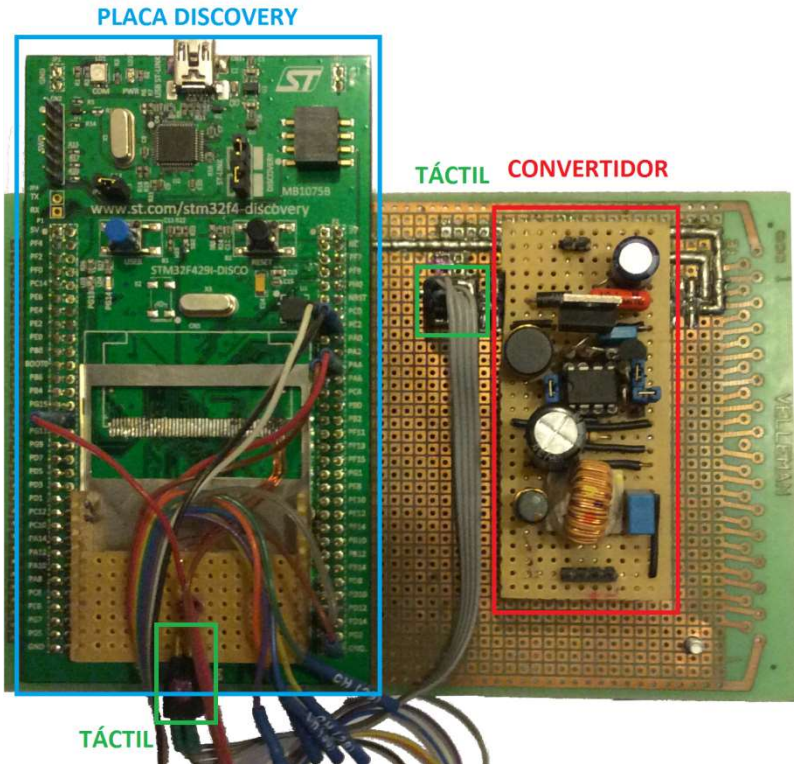


Ilustración 3.17: Montaje de la DSCO y convertidor en la placa base

A parte de apreciarse la apariencia del montaje, en esta ilustración queda patente la correcta posición de la placa DISCO en la inserción en la placa base, así como la del convertidor y el cable de la pantalla táctil.

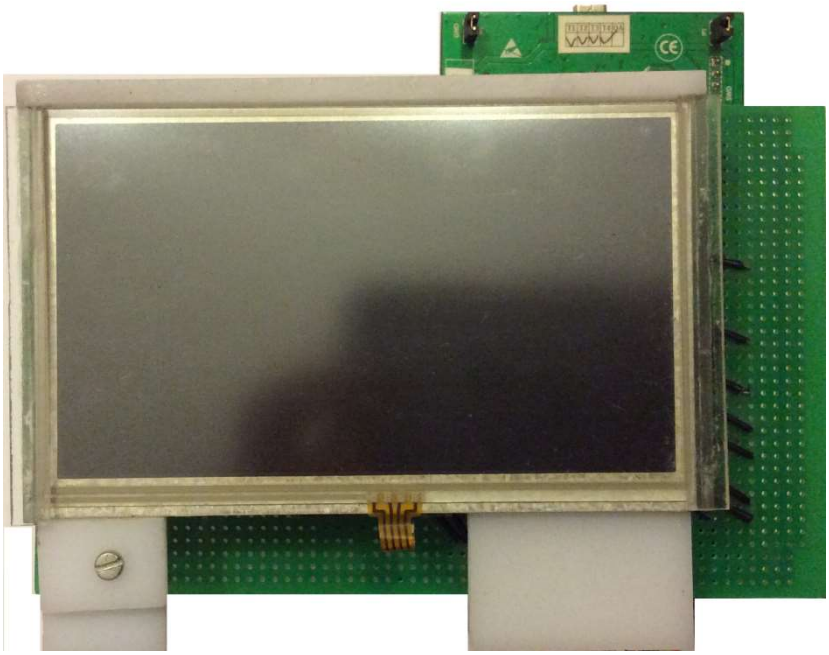


Ilustración 3.18: Montaje del display en la placa base

En esta otra ilustración aparece la cara donde va acoplado el display.

3.5 Archivo de configuración BoardConfiguration.cpp

Para la inicialización del hardware —de la placa utilizada— así como del software —la librería gráfica en sí misma—, *TouchGFX* utiliza el archivo *BoardConfiguration.cpp*. No es obligatorio que este archivo se llame así, únicamente se ha de incluir en el proyecto del entrono de desarrollo y proveer un archivo de cabecera —en este caso *BoardConfiguration.hpp*— que contenga las declaraciones externas de las funciones de configuración de hardware y software, para que puedan ser llamadas desde la función *main* —previa inclusión de este archivo de cabecera en el archivo *main.cpp*—. La función encargada de inicializar el hardware que usará *TouchGFX*, es la función *hw_init*, mientras que la utilizada para reinicializarse, es la función *touchgfx_init*. Como sucedía con el nombre del archivo, el nombre de estas funciones no tiene que ser obligatoriamente este, la única obligación es su llamada desde la función *main*. Aun así, se mantienen estos nombres por defecto para facilidad en la identificación de los mismos. La estructura concreta de este archivo, en este proyecto, es la siguiente:

```
. . .
#include <touchgfx/hal/BoardConfiguration.hpp>
//includes necesarios para el código utilizado
. . .

/**********************************************************************/
/******        Funciones de inicialización de hardware         ******/
/**********************************************************************/
//función de inicialización del periférico interno del micro LTDC [LCD_Config()]

//función de configuración de los pines del micro para el LTDC [LCD_AF_GPIOConfig()]

namespace touchgfx
{
    /**********************************************************************/
    /*****               Creación del driver táctil                  *****/
    /**********************************************************************/
    //definición de la clase que representa el driver táctil


    /**********************************************************************/
    /*****         Instancias de los driver de la capa HAL           *****/
    /**********************************************************************/
    //instancias de los drivers DMA, controlador táctil y display
    //instancia de la clase instrumentación del micro


    /**********************************************************************/
    /*****    Función de inicialización de del entrono TouchGFX     *****/
    /**********************************************************************/
    void touchgfx_init()
    {
        //iniciar la capa HAL
        //iniciar capacidades de instrumentación del micro en la capa HAL
    }


    /**********************************************************************/
    /*** Función de inicialización del hardware utilizado en TouchGFX ***/
    /**********************************************************************/
    void hw_init()
    {
        //llamada a funciones de inicialización de hardware
        //habilitar la capa1 del LTDC y el propio LTDC
        //inicializar la clase de comprobación de señales internas
        //habilitar el periférico DMA2D
    }
}
```

Código 3.20: Esquema estructural del archivo de configuración de TouchGFX BoardConfig.cpp

Ya que *TouchGFX* soporta la placa utilizada en este proyecto —STM32F429I-DISCO—, se reutiliza el archivo de configuración para la misma, con algunas modificaciones —como el tamaño de la nueva pantalla, la creación de un nuevo driver táctil o la especificación de los tamaños de los buffers gráficos—.

Lo primero que aparece en este archivo, son las sentencias de inclusión de archivos de cabecera necesarios para llamar a las funciones utilizadas, especialmente el archivo de cabecera *BoardConfiguration.hpp* —donde se declaran externamente las funciones de inicialización *hw_init* y *touchgfx_init*—. A continuación se encuentra las definiciones de varias funciones de inicialización hardware —para inicializar los pines de los puertos GPIO como salida del periférico interno LTDC, inicializar el propio LTDC o la SDRAM encargada de contener los búferes de pantalla—. Todas estas funciones no serán llamadas directamente, sino que lo serán a través de la llamada a *hw_init* —línea 41— en la función *main*. Luego, aparece un bloque delimitado por el espacio de nombres denominado *touchgfx*. Se recuerda que un espacio de nombres es un mecanismo, de C++ y de otros lenguajes orientados a objeto, que delimita una «área lógica» donde los nombres tienen un identificador único que no se repite, pero este identificador puede repetirse fuera de este espacio de nombres —repetirse de forma global o dentro de otro espacio de nombres—. Esto asegura la no repetición de nombres en proyectos grandes. Para crear o ampliar los espacios de nombres con nuevos nombres —de variables u objetos— se ha de especificar primero la palabra reservada «*namespace*» y a continuación el identificador del espacio. Para usar un objeto/variable de un espacio de nombres, se debe especificar el nombre de espacio, el operador de resolución de alcance —::— y el identificador de ese objeto/variable. Otra alternativa, sobre todo cuando se van a utilizar varios objetos/variables de un mismo espacio de nombres, es utilizar la palabra reservada «*using*» seguida del identificador del espacio de nombres y entre llaves todos los objetos/variables a utilizar, pertenecientes a este espacio. TouchGFX tiene el espacio de nombres «*touchgfx*», donde define todas las variables y clases pertenecientes a la librería. Para que una función, clase u objeto pueda hacer referencia a los elementos del espacio de nombres «*touchgfx*», puede emplearse los mecanismos antes mencionados o pertenecer, directamente, a este espacio de nombres. Esto es los que sucede en el archivo *BoardConfiguration.cpp* —líneas entre la 13 a la 48 del código anterior—, con las funciones *touchgfx_init*, *hw_init*, las instancias de los drivers del DMA, display y táctil, así como la creación del driver táctil —línea 18 del código anterior—. Esto permite que todos estos elementos, al hacer referencia a otros elementos de la librería *TouchGFX*, los llamen sin más —sin tener que especificar el espacio de nombres «*touchgfx*»—.

En el caso de este proyecto, donde se ha instalado un nuevo display en la placa STM32F429I-DISCO, ha sido necesario crear un nuevo driver táctil —explicado en las siguientes secciones— y se ha decidido que la clase que representa este driver, se localice dentro del archivo BoardConfiguration.cpp, aunque no es lo habitual. Lo habitual es localizarlo en un archivo aparte, e incluir su cabecera en el archivo de configuración para poder crear el objeto o instancia de este driver, para posteriormente agregarlo al objeto que representa la capa HAL. En las secciones siguientes se detallará cada una de las partes en las que se divide este archivo, así como el funcionamiento de la pantalla táctil y su control por parte del periférico externo STMPE811 de la placa.

3.5.1 Inicialización Hardware.

La inicialización del hardware, que TouchGFX necesita, se realiza en la función *hw_init* del archivo de configuración BoarConfiguration.cpp. Su contenido es el siguiente:

```
1   . . .
2   /**********************************************************************/
3   /*** Función de inicialización del hardware utilizado en TouchGFX ***/
4   /**********************************************************************/
5
6   void hw_init()
7   {
8       /* Configura el LCD */
9       LCD_Config();
10
11      /* Habilita la capa 1 */
12      LTDC_LayerCmd(LTDC_Layer1, ENABLE);
13
14      /* recarga la configuración del LTDC de forma inmediata
15        (sin esperar al sincronismo vertical) */
16      LTDC_ReloadConfig(LTDC_IMReload);
17
18      /* Habilita el LCD */
19      LTDC_Cmd(ENABLE);
20
21      /* Inicializa la clase GPIO de depuración */
22      GPIO::init();
23
24      /* Habilitación del periférico DMA2D */
25      RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_DMA2D, ENABLE);
26  }
```

Código 3.21: Función de inicialización de hardware dentro del archivo BoardConfiguratio.cpp

Esta es la función de inicialización del hardware que prácticamente viene tal cual en el archivo BoardConfiguration.cpp para la placa Discovery con su display original. Ello es debido a que en dicha función, directamente, no se especifican las características del display usado, aunque si se hace en la función *LCD_config* que es llamada. Esta función sí es modificada para adecuarse a los parámetros del nuevo display instalado, y es la encargada de configurar el periférico interno LTDC que controlará el nuevo display instalado

Siguiendo la inicialización hardware de la función *hw_init*, lo que se hace a continuación es habilitar la capa 1 del periférico LTDC —línea 12—. Este periférico dispone de dos capas y un fondo de color, quedando la capa 2 sobre la capa 1 y esta sobre el fondo. Por tanto, la capa 2 es la de primer plano y la capa 1 es las de fondo [4]:

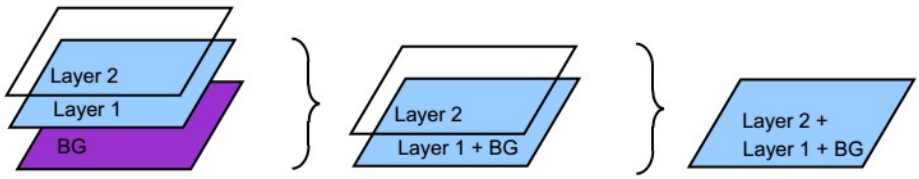


Ilustración 3.19: Esquema de solapamiento de capas del periférico LTDC

Ambas capas y el fondo de color son mezclados mediante la configuración del canal alfa de cada capa. Para *TouchGFX* esta funcionalidad no va a ser utilizada, solo se habilitará la capa de fondo, la capa 1, y tanto el fondo de color como la capa de primer plano –la capa 2 — permanecen inhabilitadas por defecto.

Luego se instruye al periférico LTDC que cargue toda la configuración establecida de forma inmediata, sin esperar al siguiente sincronismo vertical —línea 16—.Finalmente, el periférico LTDC es habilitado en la línea 19.

Lo último que se hace en esta función de inicialización hardware, es poner en marcha la clase de depuración GPIO —que saca al exterior señales internas de funcionamiento del entorno gráfico, como la señal de actualización del display (tick)—, y habilita el periférico *DMA2D* utilizado por el entrono gráfico para copiar bloques con representación bidimensional de memoria en el buffer secundario de pantalla. La configuración de este último periférico no se hace en este archivo, ya que esta configuración va a cambiar según los requerimientos que la librería gráfica *TouchGFX* tenga en cada momento. Esto se hace de forma interna y no se tiene acceso al código. Sin embargo, en un componente desarrollado para el proyecto —la clase *Graph*—, se utiliza directamente el periférico *DMA2D*, asegurándose que la librería no lo va a poder utilizar en el mismo momento que lo haga esta clase.

A la vista de lo expuesto, el paso de configuración hardware donde se concentra mayor trabajo, es en la llamada a la función la *LCD_config*. La estructura de esta función es la siguiente:

```
1  static void LCD_Config(void)
2  {
3
4      /* Habilita el reloj del LCD */
5
6      /* Configura los pines del micro para señales de datos y control del LCD
7         (llamada a  LCD_AF_GPIOConfig())  */
8
9      /* Configurar la SDRAM externa (llamada a SDRAM_Init()) */
10
11     /* Configurar el reloj de pixeles del periférico LTDC */
12
13     /* Configurar el periférico LTDC */
14 }
```

Código 3.22: Esquema de la función de configuración del LCD

Los principales pasos de configuración de esta función son, la habilitación del reloj del periférico interno LTDC, configurar los pines oportunos del micro para que se disponga de las señales de datos y control de este periférico al exterior, inicializar la memoria SDRAM externa, configuración y habilitación del reloj de datos del display y configuración del periférico interno LTDC. La estructura de bloque del periférico LTDC es la siguiente [4]:

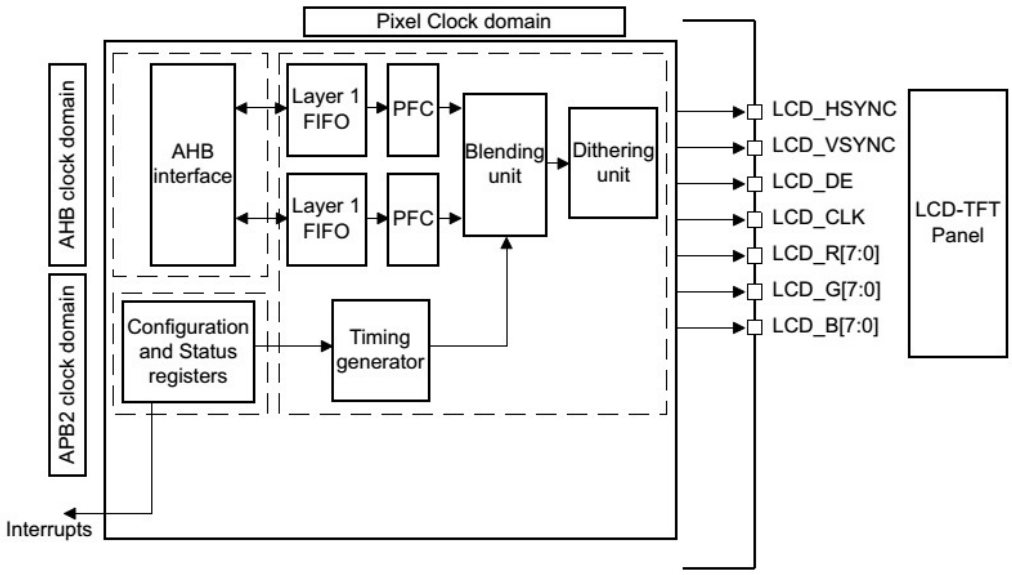


Ilustración 3.20: Estructura del periférico LTDC

Consta de tres dominios de reloj: AHB clock, encargado de gobernar la transferencia de datos de memoria a cada una de las capas; APB2 clock, encargado de gobernar la transferencia de datos a registros de configuración, y Pixel clock, encargado de gobernar transferencia de puntos a la pantalla y generar las señales de sincronización. Cuando se habilita el reloj del LCD en la función *LCD_Config* —línea 4 del Código 3.22—, se está habilitando el APB2 clock. En cambio, cuando se configura el reloj de pixeles del periférico LTDC —línea 11 del Código 3.22— se está configurando Pixel clock. En el esquema de este periférico se aprecian las dos capas —Layer1 y Layer2— con sus correspondientes convertidores de formato de puntos –PFC— y la unidad de mezclado de capas —Blending unit—, en el final del camino de la señal está el bloque de difuminado —Dithering unit—, especialmente útil para formato de puntos de 16 bits —RGB565—. Pero para el uso de TouchGFX solo se usa la capa 1, como ya ha sido comentado. Las señales que se ofrecen al exterior son las referentes al color del punto en cada momento —LCD_R, LCD_G y LCD_B—, el reloj de transferencia de puntos a pantalla —LCD_CLK— y las señales de sincronización —LCD_HSYNC, LCD_VSYNC y LCD_DE—. La configuración del los pines del micro, para que estás señales estén presente al exterior, la configuración del reloj de pixeles —Pixel clock— así como la configuración del propio periférico, son objeto de las siguientes secciones. Todos estos pasos de configuración, junto con la inicialización de la SDRAM, son los puestos en marcha por la función *LCD_Config* —Código 3.22—.

3.5.1.1 Configuración de los pines del micro para el display.

La configuración de los pines del micro, para suministrar las señales del periférico interno LTDC, se realiza mediante la función *LCD_AF_GPIOConfig*, que es llamada por la función *LCD_Config*, que a su vez es llamada por la función del *hw_init*, perteneciente al archivo de configuración BoardConfiguration.cpp —ver inicio de la sección 3.5—. Esta función de configuración de pines está inspirada en la homónima utilizada en la librería de la placa STM32F429I-DISCO, con la diferencia que en la utilizada aquí, no se configuran los pines que ofrecen al exterior las señales de sincronismo horizontal y vertical para controlar el display, ya que el display utilizado no lo necesita y los pines correspondientes pueden ser reutilizados para otras funcionalidades.

Cada señal del LTDC tiene asignado una serie de pines determinados de un puerto específico del micro, lo que significa que no se puede configurar cualquier pin del micro para que suministre estas señales. Aunque sí es cierto que una misma función alternativa puede estar presente en diferentes pines del micro, el pin y puerto quedan determinados por los ya utilizados para el display de 2,4 pulgadas que viene instalado en la placa Discovery.

Para saber cuáles son estos pines, se puede consultar las tablas 10 y 12 del datasheet del micro STM32F429 [5] y consultar, en el manual de usuario de la placa Discovery, UM1670 [6] la tabla 6 o el esquema 5. De esta información se extrae que los puertos a tratar son GPIO A, B, C, D, F y G. En cada uno de estos puertos serán configurados los pines que suministran las señales del periférico LTDC al exterior, ya utilizados en la placa, y que se resumen en la siguiente tabla:

Señal LTDC		Puerto (GPIO)	Pin del Puerto	Señal LTDC		Puerto (GPIO)	Pin del Puerto
ROJO	R2 (LSB)[16]	C	10	AZUL	B2 (LSB)[16]	D	6
	R3	B	0		B3	G	11
	R4	A	11		B4	G	12
	R5	A	12		B5	A	3
	R6	B	1		B6	B	8
	R7 (MSB)[17]	G	6		B7 (MSB)[17]	B	9
VERDE	G2 (LSB)[16]	A	6				
	G3	G	10				
	G4	B	10	CONTROL	CLK	G	7
	G5	B	11		DE	F	10
	G6	C	07				
	G7 (MSB)[17]	D	03				

Tabla 14 : Puertos y pines asignados a las señales del periférico LTDC

El código resumido de la función *LCD_AF_GPIOConfig* es el siguiente:

```
1  static void LCD_AF_GPIOConfig(void)
2  {
3      GPIO_InitTypeDef GPIO_InitStruct;
4
5      /* Habilitación de los relojes AHB GPIOA, GPIOB, GPIOC, GPIOD, GPIOF y GPIOG */
6      RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA | RCC_AHB1Periph_GPIOB | \
7                             RCC_AHB1Periph_GPIOC | RCC_AHB1Periph_GPIOD | \
8                             RCC_AHB1Periph_GPIOF | RCC_AHB1Periph_GPIOG, ENABLE);
9      /* Configuración GPIOA */
10     GPIO_PinAFConfig(GPIOA, GPIO_PinSource3, GPIO_AF_LTDC);
11     GPIO_PinAFConfig(GPIOA, GPIO_PinSource6, GPIO_AF_LTDC);
12     GPIO_PinAFConfig(GPIOA, GPIO_PinSource11, GPIO_AF_LTDC);
13     GPIO_PinAFConfig(GPIOA, GPIO_PinSource12, GPIO_AF_LTDC);
14
15     GPIO_InitStruct.GPIO_Pin = GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_6|GPIO_Pin_11 | GPIO_Pin_12;
16     GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
17     GPIO_InitStruct.GPIO_Mode = GPIO_Mode_AF;
18     GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
19     GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
20     GPIO_Init(GPIOA, &GPIO_InitStruct);
21     /* Configuración GPIOB */
22     GPIO_PinAFConfig(GPIOB, GPIO_PinSource0, 0x09);
23     GPIO_PinAFConfig(GPIOB, GPIO_PinSource1, 0x09);
24     GPIO_PinAFConfig(GPIOB, GPIO_PinSource8, GPIO_AF_LTDC);
25     GPIO_PinAFConfig(GPIOB, GPIO_PinSource9, GPIO_AF_LTDC);
26     GPIO_PinAFConfig(GPIOB, GPIO_PinSource10, GPIO_AF_LTDC);
27     GPIO_PinAFConfig(GPIOB, GPIO_PinSource11, GPIO_AF_LTDC);
28     GPIO_InitStruct.GPIO_Pin = GPIO_Pin_0 | GPIO_Pin_1 | GPIO_Pin_8 | \
29                                GPIO_Pin_9 | GPIO_Pin_10 | GPIO_Pin_11;
30     GPIO_Init(GPIOB, &GPIO_InitStruct);
31     /* Configuración GPIOC */
32                               //configuración de los pines 7 y 10
33     /* Configuración GPIOD */
34                               //configuración de los pines 3 y 6
35     /* Configuración GPIOF */
36                               //configuración del pin 10
37     /* Configuración GPIOG */
38                               //configuración de los pines 6, 7, 10, 11 y 12
39 }
```

Código 3.23: Función resumida de la configuración de pines para el periférico LTDC

[16] Least Significant Bit —bit menos significativo—

[17] Most Significant Bit —bit más significativo—

Inicialmente se declara una variable de tipo estructura *GPIO_InitTypeDef* —variable llamada *GPIO_InitStruct*, línea 3— que va a contener parámetros de configuración de los puertos y será reutilizada para la inicialización de cada uno de ellos. A continuación se habilitan los relojes de todos los puertos a configurar —líneas de la 6 a la 8—. A partir de aquí, la configuración se repite para cada puerto. Primero se selecciona la salida alternativa LTDC en el multiplexor de funciones alternativas del micro para los pines del puerto tratado [4]:

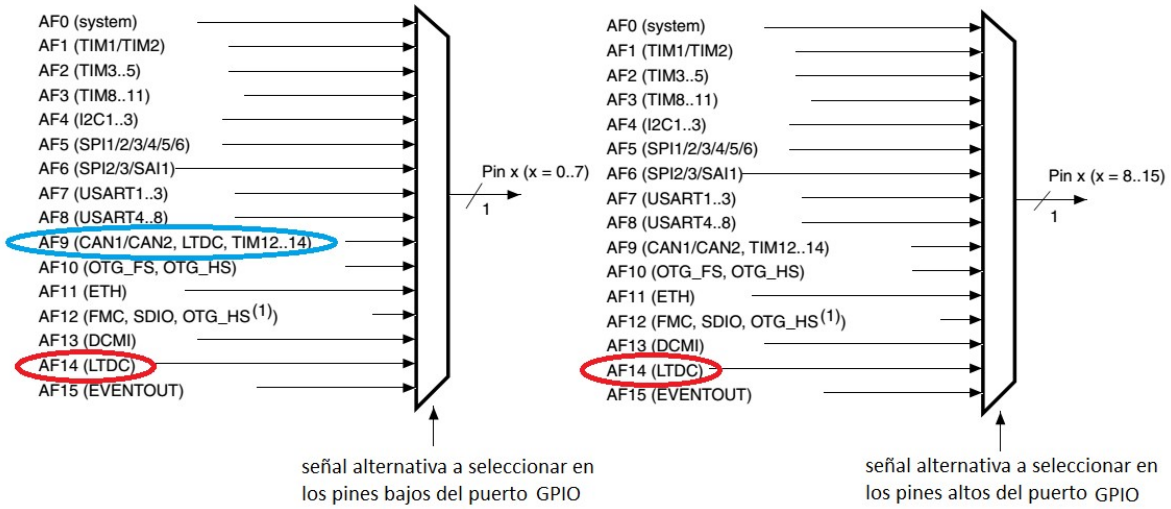


Ilustración 3.21: Señales de funciones alternativas para los pines del micro STM32F429

Como se aprecia en esta figura, hay dos tipos de multiplexores de funciones alternativas, uno para los pines bajos del puerto —pines del 0 al 7– y otro para los altos —del 8 al 15—. Para cada tipo de multiplexor la función 14 —AF14— es la referente a una señal del periférico LTDC, por ello la librería estándar de periféricos del micro provee una constante con este valor, llamada *GPIO_AF_LTDC* –aparece en las líneas 10, 11, 12, 13, 24, 25, 26 y 27 del código anterior—. Otra función alternativa con posibles señales del LTDC es la señal AF9 —(CAN1/CAN2, LTDC,TIM12..14)— del multiplexor conectado a los pines bajos del puerto tratado. En esta función alternativa la librería no ofrece ninguna constante con un nombre significativo relacionada con este periférico. Para conocer qué señal concreta es la suministrada, tanto en la función alternativa AF14 como en la AF9, para los pines de un puerto concreto, se ha de consultar el datasheet del micro [5]. El discernimiento entre utilizar la función alternativa 14 o la 9 va a depender qué señal concreta del LTDC se quiera obtener para el pin tratado, ya que es posible que para este pin, esté disponible una señal del LTDC en la función alternativa 14 y otra distinta del LTDC en la 9. Se remite al datasheet del micro para obtener, de forma genérica, esta información. Para el caso del puerto A —GPIOA—, los cuatro pines utilizados –3, 6, 11 y 12—, obtienen sus señales, procedentes del LTDC, desde la función alternativa AF14 —líneas 10 a la 13 del código anterior—. Luego se rellena la estructura *GPIO_InitStruct* con los parámetros específicos de estos pines —como que la salida sea push-pull, no haya resistencia de pull-up o pull-down y que la rapidez de respuesta sea la máxima posible (aunque esta especificación es realmente para funcionalidad del pin distinta a la función alternativa)—. Un parámetro a resaltar, en el relleno de esta estructura, es la especificación que el pin utilizará la función alternativa —línea 17—, algo que parece redundante. Sin embargo no lo es, cuando se selecciona la función alternativa concreta en los multiplexores correspondientes —antes comentados—, no se está especificando que esta funcionalidad llegue al pin; el pin puede configurarse como entrada/salida digital, entrada/salida analógica —para el caso de adquisición de datos por ADC o salida de señales desde el DAC, respectivamente—, o para que use la función alternativa previamente seleccionada. Esta selección constituye el multiplexor de salida del pin. Una vez especificados todos los parámetros de configuración para los pines de un puerto, son inicializados mediante la llamada a la función de la librería de periféricos estándar de micro llamada *GPIO_Init*, que recibe como argumentos el puerto al que configurar los pines específicos y la estructura con la información de configuración para ellos —línea 20 para la configuración de los pines del puerto A—. Para la configuración de los pines del puerto B —líneas de la 22 a la 30—, si sigue el mismo proceso, con la diferencia que dos de sus pines —0 y 1—,

obtienen las respectivas señales del periférico LTDC desde la función alternativa AF9 —líneas 22 y 23—. También hay que destacar que no es necesario volver a rellenar la estructura de configuración, ya que los valores que contiene van a ser comunes en la inicialización del resto de puertos. Por último, destacar que los pines 10 y 12 del puerto G, van a obtener las señales desde la función alternativa AF9 —código no mostrado—.

3.5.1.2 Inicialización de la RAM externa.

El siguiente paso que se realiza en la función *LCD_Config* — perteneciente al archivo de configuración *BoardConfiguration.cpp* —, después de configurar ciertos pines del micro como salidas del periférico interno LTDC, es configurar las RAM externa al micro con esta porción de código:

```
1    /* Configura el periférico interno del micro FMC para usar la SDRAM externa
2      (donde estarán localizados los búferes de pantalla) */
3    SDRAM_Init();
```

Código 3.24: Configuración de la RAM externa dentro de la función LDC_Config

Se trata de la simple llamada a la función *SDRAM_Inint* contenida en el archivo *stm32f429i_discovery_sdram.cpp* que pertenece a la librería de la placa STM32F429I-DISCO. Esta función, internamente va a realizar tres tareas: configurar los pines del micro, necesario para dar salida a los buses de dirección, de datos y de control para la comunicación con la RAM externa; configurar el periférico interno FMC —controlador de memoria flexible— para mapear esta memoria dentro del espacio de direccionamiento del micro y generar las señales de los buses antes mencionadas y finalmente realizar la secuencia de inicialización de la RAM.

Como sucedía para el caso de la configuración alternativa de los pines del micro para dar servicio hacia el exterior del periférico interno LTCD, para el FMC se tendrán ciertos pines del micro que podrán sacar sus señales al exterior. Pero la restricción mayor es el ruteado de las pistas en la placa Discovery que impiden que la configuración pueda ser diferente a la que hace la función de la librería *SDRAM_Inint* [6]:

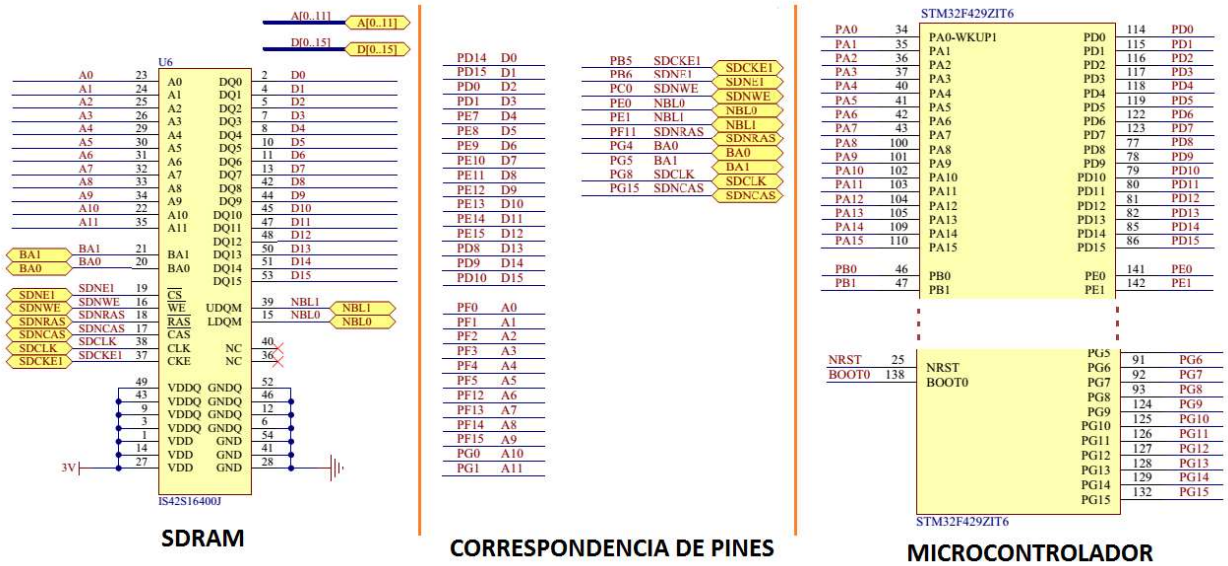


Ilustración 3.22: Esquema de la SDRAM y la correspondencia de señales con pines del micro

En esta ilustración aparecen, de forma simplificada, el esquema del micro —derecha—, el de la SDRAM —izquierda— y la correspondencia de las señales de los buses de la RAM con los pines utilizados en el micro en la placa Discovery —centro—. Esta parte central de la ilustración informa de los pines del micro que la función *SDRAM_Inint* va a configurar y que, claramente, no pueden ser otros aunque dichas señales puedan estar potencialmente presentes en otros pines del micro. Para ello, la función *SDRAM_Inint* llama a otra función de la librería de la placa, *SDRAM_GPIOConfig* —contenida en el mismo archivo *stm32f429i_discovery_sdram.cpp*— cuyo esqueleto es el siguiente:

```
void SDRAM_GPIOConfig(void)
{
 . . .
/*-- GPIOs Configuration -----------------------------------------------------*/
/*
 +-------------------+-------------------+-------------------+-------------------+
 +                       SDRAM pins assignment                                   +
 +-------------------+-------------------+-------------------+-------------------+
 | PD0  <-> FMC_D2   | PE0  <-> FMC_NBL0  | PF0  <-> FMC_A0    | PG0  <-> FMC_A10   |
 | PD1  <-> FMC_D3   | PE1  <-> FMC_NBL1  | PF1  <-> FMC_A1    | PG1  <-> FMC_A11   |
 | PD8  <-> FMC_D13  | PE7  <-> FMC_D4    | PF2  <-> FMC_A2    | PG8  <-> FMC_SDCLK |
 | PD9  <-> FMC_D14  | PE8  <-> FMC_D5    | PF3  <-> FMC_A3    | PG15 <-> FMC_NCAS  |
 | PD10 <-> FMC_D15  | PE9  <-> FMC_D6    | PF4  <-> FMC_A4    |-------------------+
 | PD14 <-> FMC_D0   | PE10 <-> FMC_D7    | PF5  <-> FMC_A5    |
 | PD15 <-> FMC_D1   | PE11 <-> FMC_D8    | PF11 <-> FMC_NRAS  |
 +-------------------| PE12 <-> FMC_D9    | PF12 <-> FMC_A6    |
                     | PE13 <-> FMC_D10   | PF13 <-> FMC_A7    |
                     | PE14 <-> FMC_D11   | PF14 <-> FMC_A8    |
                     | PE15 <-> FMC_D12   | PF15 <-> FMC_A9    |
 +-------------------+-------------------+-------------------+
 | PB5 <-> FMC_SDCKE1|
 | PB6 <-> FMC_SDNE1 |
 | PC0 <-> FMC_SDNWE |
 +-------------------+

*/

 /* Common GPIO configuration */

. . .

}
```

Código 3.25: Esqueleto de la función que configura los pines del micro para acceder a la RAM externa

En este esqueleto se muestra el comentario que informa de la configuración que se va a realizar en cada uno de los pines. A continuación se procede a configurar cada uno de los puertos del micro que contienen estos pines —no mostrado en el código anterior—. No se va a entrar en más detalle ya que es código perteneciente a la librería de la placa, que no ha sido desarrollado para el proyecto o modificado para adaptarlo, tan solo ha sido reutilizado.

A continuación, la función *SDRAM_Init* configura el periférico interno FMC con sentencias de código directamente en su cuerpo. El parámetro de configuración que más es interesante conocer, es la dirección de inicio en la que se localiza la memoria RAM, dentro del espacio de direcciones. Las posibles áreas de de direccionamiento del periférico FMC, para distintos tipos de memorias es [4]:

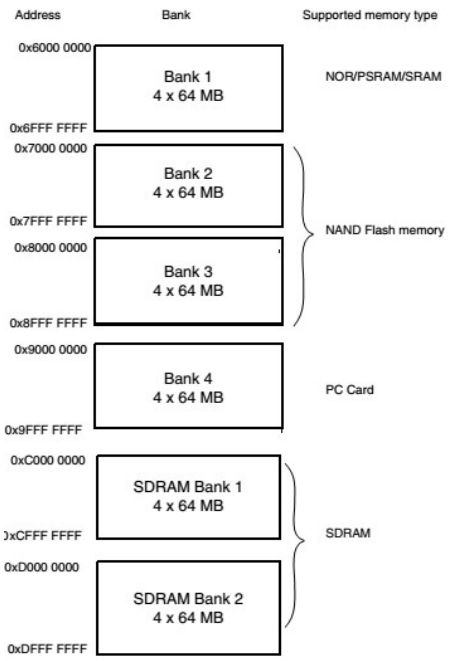


Ilustración 3.23: Espacio de direccionamiento del periférico FMC

El FMC divide la zona de direccionamiento en diferentes bancos cada uno de 256MB. Los bancos de interés son los dedicados a la RAM: SDRAM Bank1 y SDRAM Bank2. La función *SDRAM_Init* localiza la RAM externa dentro del banco SDRAM Bank2, con lo que su dirección de inicio será 0xD0000000. Esta será la dirección donde se localice el buffer gráfico y cuyo valor será almacenado en la variable global —al archivo de configuración *Boardconfiguration.cpp*— *frameBuf0*, utilizada para la configuración de la capa 1 del LCD —ver sección 3.5.14— y para la inicialización software de TouchGFX —ver sección 3.5.2— Como la memoria utiliza es de 64Mbits —o 8 MB—, la última dirección de la RAM es 0xD07FFFFF —8 MB en hexadecimal es 0x00800000—. La resolución del display a utilizar es de 272x480 y la profundidad de color usada será de 16 bits —RGB565—, por tanto, se necesita un tamaño de buffer gráfico, expresado en bytes de 272x480x2 = 261120 bytes o 255 KB. Si se desea tener un buffer secundario de memoria y uno adicional, para animaciones, el mapa de memoria del banco 2 para la memoria RAM del FMC queda:

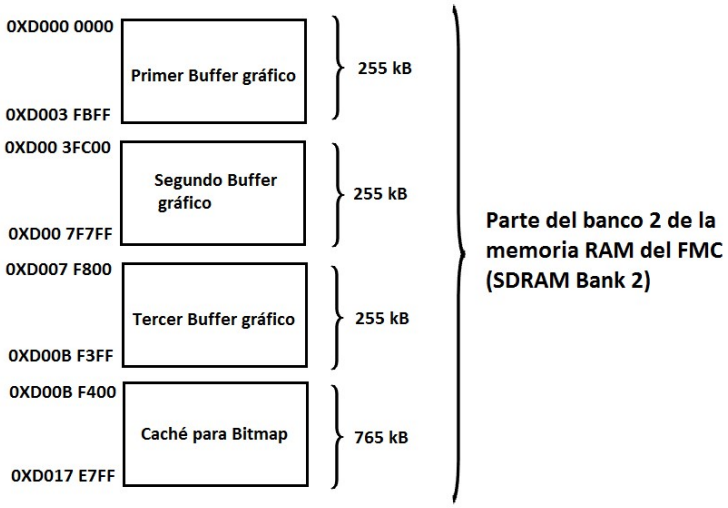


Ilustración 3.24: Mapa de la memoria RAM externa utilizada por TouchGFX

TouchGFX utiliza, de forma necesaria, el primer y segundo buffer gráfico. El tercero puede ser utilizado para transiciones entre distintas pantallas —no utilizado en este proyecto—. El último bloque que aparece en la figura —de 765 kB— está destinado a albergar los mapas de bits en vez de que se alojen en la memoria flash interna. La ventaja principal es la posible mayor capacidad —este bloque tiene un tamaño pequeño, pero la memoria RAM aun dispone de 6,5 MB para poder ser utilizada—. Este último bloque no es utilizado en este proyecto y se deja a disposición de desarrollos futuros.

Esta división de memoria se realiza en la inicialización software de TouchGFX explicada en la sección 3.5.2.

Finalmente, en la función de inicialización *SDRAM_Inint* realiza una secuencia de inicialización de la RAM, que puede consultarse directamente en su código fuente y que corresponde con los pasos de inicialización expuestos en el apartado 37.7.3 en el manual de referencia RM0090.

3.5.1.3 Configuración del reloj del display.

El reloj del LTDC —el que controla la sincronización de las señales que este periférico manda al exterior— se extrae de un PLL[18] interno del micro que es también utilizado para disciplinar el periférico interno SAI —serial audio interface—, de ahí que su denominación sea PLLSAI. El esquema resumido de este PLL, así como otros elementos relacionados, se muestran en la siguiente figura [4]:

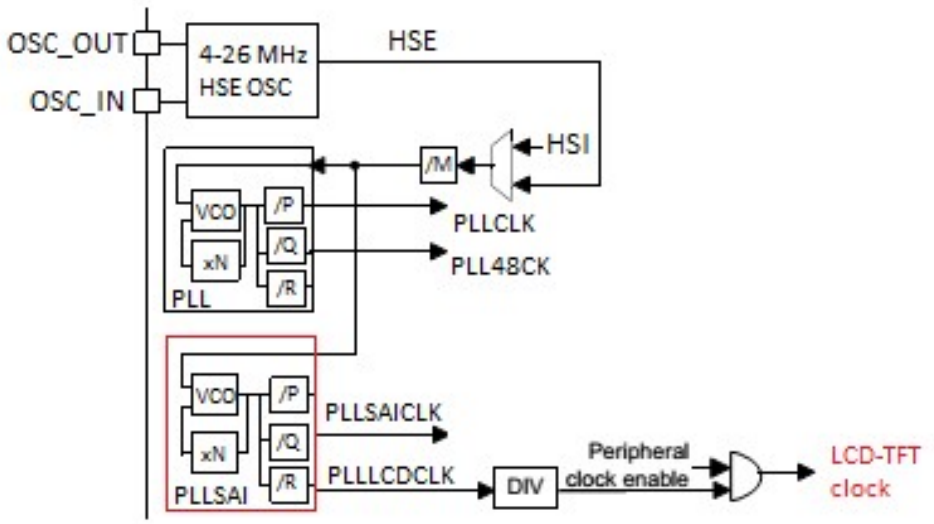


Ilustración 3.25: Esquema de bloques del reloj del LTDC

Originariamente, la señal del reloj del LTDC procede del reloj conectado externamente al micro entre los terminales OSC_OUT y OSC_IN, que es de 8MHz. Este reloj —HSE— llega hasta un divisor de frecuencia —/M—, cuyo valor es ocho y está fijado en el archivo *system_stm32f4xx.c* —archivo encargado de inicializar el reloj del sistema, señal de reloj no mostrada en este esquema—. Por tanto, la señal que se tiene a la salida de este divisor es de 1MHz. A partir de aquí la señal se bifurca en dos, por un lado llega al PLL principal, donde se generará el reloj precursor del sistema, y por otro, al PPLSAI, donde se generarán los relojes del LTDC y del SAI. Para ello, el PLLSAI consta de un multiplicador de frecuencia —N— y dos divisores de frecuencia —Q y R—. Para el reloj del LTDC, además hay un divisor a la salida del PLLSAI. La configuración del reloj del LTDC consistirá en especificar los valores de N y R del PLLSAI y el del divisor a su salida. Por tanto, en la función *LCD_Config*, que es donde se

[18] Phase Locked Loop —lazo de seguimiento de fase—

realiza la configuración del reloj de puntos del LTDC, después de haber configurado los pines del micro como salida del LTDC e inicializar la RAM externa, se establece la siguiente porción de código:

```
1   /* Habilita el reloj de Pixeles -----------------------------------------------------*/
2   /* Configura el preescalador PLLSAI para el LCD */
3   /* Entrada PLLSAI_VCO = HSE_VALUE/PLL_M = 1 Mhz */
4   /* Salida PLLSAI_VCO  = Entrada PLLSAI_VCO * PLLSAI_N = 216 Mhz */
5   /* PLLLCDCLK = Salida PLLSAI_VCO/PLLSAI_R = 216/3 = 72 Mhz */
6   /* Frecuencia de reloj LTDC = PLLLCDCLK / RCC_PLLSAIDivR = 72/8 = 9 Mhz */
7   RCC_PLLSAIConfig(216, 7, 3);
8   RCC_LTDCCLKDivConfig(RCC_PLLSAIDivR_Div8);
9
10
11  /* Habilitación del reloj PLLSAI */
12  RCC_PLLSAICmd(ENABLE);
13  /* Esperar a la habilitación del PLLSAI */
14  while (RCC_GetFlagStatus(RCC_FLAG_PLLSAIRDY) == RESET)
15  {
16  }
```

Código 3.26: Configuración Pixel clock del LTDC en la función LCD_Config

Para configurar los valores del PLLSAI, la librería estándar de periféricos del micro STM32F429 provee la función *RCC_PLLSAIConfig*. Esta función tiene como argumentos los valores de N, Q y R en este orden. Aunque no se va a utilizar el periférico SAI, debido a la estructura de esta función, se debe especificar el valor de Q. Este valor ha de estar entre 2 y 15; se da un valor arbitrario de 7. La frecuencia de envío de puntos a pantalla ha de ser de 9MHz —tal y como se mostró en la sección 3.3 Tabla 10—, así que, se ha de establecer un valor multiplicador —N— y dos valores divisores —R y el divisor exterior al PLLSAI— que operados con el valor de 1MHz —frecuencia a la entrada del PLLSAI— se obtenga 9MHz. El valor de N ha de oscilar entre 192 y 432, el de R entre 2 y 7 y el valor del divisor exterior puede ser 2, 4, 8 y 16. Una combinación de valores permitidos, con los que se obtiene el valor del reloj deseado, son: 216 para N, 3 para R y 8 para el divisor externo. La configuración del PLLSAI se realiza en la línea 7 del código mostrado, mientras que la del divisor exterior se realiza en la 8. Luego se habilita el PLL, ya que inicialmente está inhabilitado y no está permitido configurarlo una vez esté habilitado. Por último se espera, en el bucle que aparece entre la líneas 14 a la 16, que el PLLSAI está preparado.

3.5.1.4 Configuración del periférico LTDC.

Como ya ha sido comentado, el periférico interno del micro, LTDC, es el encargado de controlar el nuevo display instalado en la placa Discovery, y su esquema de bloques ha sido mostrado al inicio de la sección 3.5.1. Después de la configuración del reloj de pixeles, el siguiente paso de configuración, dentro de la función *LCD_Config*, es la del LTDC en sí mismo. Esto significa, en primer lugar, establecer los parámetros característicos de las señales control para que se pueda mandar información a cualquier punto de pantalla; en segundo lugar, especificar los parámetros que determinan las dimensiones de las capas usadas —se mostró un esquema de las capas al inicio de la sección 3.5.1— así como su esquema de color, y en tercer lugar, habilitar características adicionales, como en este caso, el difuminado.

A pesar que el display instalado no necesita las señales de sincronización horizontal y vertical del periférico LTDC —LCD_HSYNC y LCD_VSYNC—, es necesario configurarlas debido a que la señal de habilitación de datos —LTDC_DE—, necesaria para este display, es derivada de la configuración de las dos anteriores. Lo primero es configurar la polaridad de estas señales junto con la del reloj de pixeles. Esto se lleva a cabo mediante la siguiente porción de código de la función *LCD_Config*, del archivo *BoardConfiuratin.cpp*:

```
1   /* Defines para hacer los cambios de configuración más sencillos ------*/
2   #define H_SYN_W 41
3   #define V_SYN_W 10
4   #define H_BACK_PORT 2
5   #define V_BACK_PORT 2
6   #define WITH_ACTIVO 480
7   #define HEIGHT_ACTIVO 272
8   #define H_FROM_PORT 2
9   #define V_FROM_PORT 50
10
11
12  /* Inicialización del LTDC ----------------------------------------------------*/
13
14  /* Configuración de las polaridades de las señales de control */
15  LTDC_InitStruct.LTDC_HSPolarity = LTDC_HSPolarity_AL;
16  LTDC_InitStruct.LTDC_VSPolarity = LTDC_VSPolarity_AL;
17  LTDC_InitStruct.LTDC_DEPolarity = LTDC_DEPolarity_AL;
18  LTDC_InitStruct.LTDC_PCPolarity = LTDC_PCPolarity_IPC;
```

Código 3.27: Configuración de la polaridad de las señales de control del LTDC

Entre las líneas 15 a 18 se configuran las polaridades de los sincronismos horizontal y vertical y de las señales de habilitación de datos y reloj de puntos, respectivamente. La polaridad de la señal de habilitación de datos es activa a nivel bajo mientras que los datos son mandados en cada pulso de reloj de puntos. Las señales de sincronismo horizontal y vertical se configuran con polaridad activas a nivel bajo ya que este es el valor habitual para este tipo de señales —aunque no van a ser utilizadas—. Entre las líneas 2 a 9, se especifican una serie de constantes cuyos valores son las distintas temporizaciones de las señales de control. Estas definiciones permiten modificar los parámetros de configuración sin más que actualizar el valor de las constantes —dejando el resto de código sin modificar—. Los valores de estas constantes de temporización son extraídos de de la hoja de datos del fabricante del modelo de display —InnoLux AT050TN33— y se pueden encontrar igualmente en la Tabla 10 de la sección 3.3 —aunque los datos de los sincronismos han de ser inferidos ya que el display utilizado no dispone de tales señales (esto será mostrado más adelante)—. La representación, en el esquema de pantalla, de estos parámetros es la siguiente [8]:

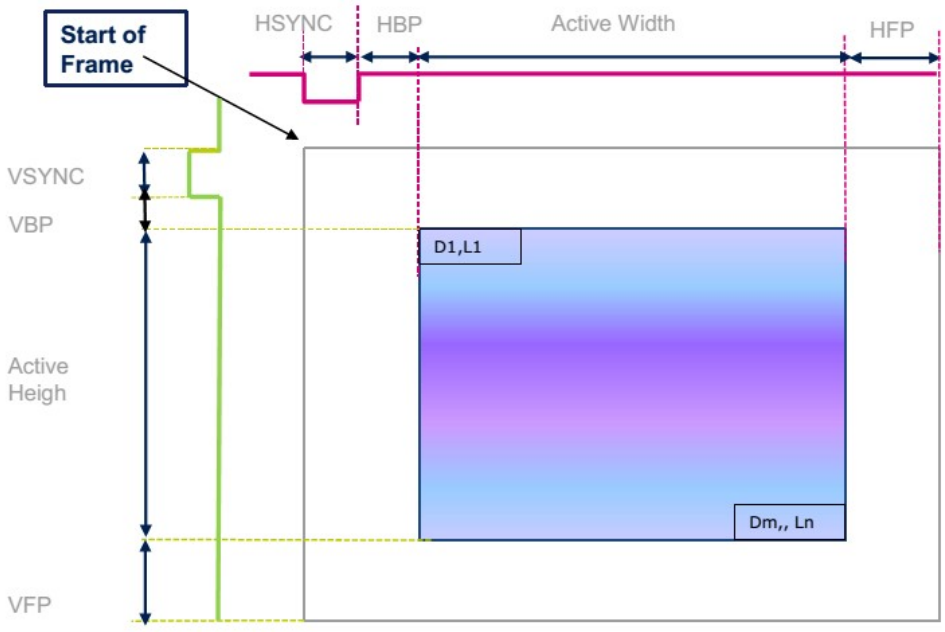


Ilustración 3.26: Parámetros de las señales de sincronización de pantalla

Las especificaciones referentes a temporizaciones horizontales, están expresadas en puntos, mientras que las verticales en líneas —esto ya ha sido expuesto en la sección 3.3, recordándose aquí para evitar confusiones—.

HSYNC es el ancho del sincronismo de línea —sincronismo horizontal—, indica a la pantalla cuando iniciar el dibujado de una nueva línea de pantalla y aparece en la línea 2 del código anterior con el valor de 41. HBP y HFP —líneas 4 y 8 del código— son los pórticos trasero y frontal, respectivamente, y sus valores son de 2 y 2. Ambas temporizaciones hacen referencia puntos de una misma línea que no son visibles. HBP es la zona anterior a la visible de una línea, mientras que HFP es la zona posterior. El ancho de área activa horizontal —Active Width—es la zona de la línea que sí es visible, y está especificado en la línea 6 del código con el valor del ancho de pantalla en puntos, es decir 480. El sincronismo vertical VSYNC, es la señal que indica al display cuando iniciar un nuevo cuadro y su valor es de 10 —línea 3 del código—. Sus pórticos trasero y frontal, VBP y VFP —con significado equivalente a los horizontales— tiene los valores respectivos de 2 y 4 en las líneas 5 y 9 del código.

Sin embargo, aunque estos valores de temporización son necesarios para la configuración del periférico LTDC, el display utilizado no especifica información de sincronismos —horizontal y vertical— ya que no dispone de ellos. Se hace necesario relacionar las temporizaciones de la señal de habilitación de datos con estos sincronismos. Para ello se va a hacer uso de las siguientes gráficas [25]:

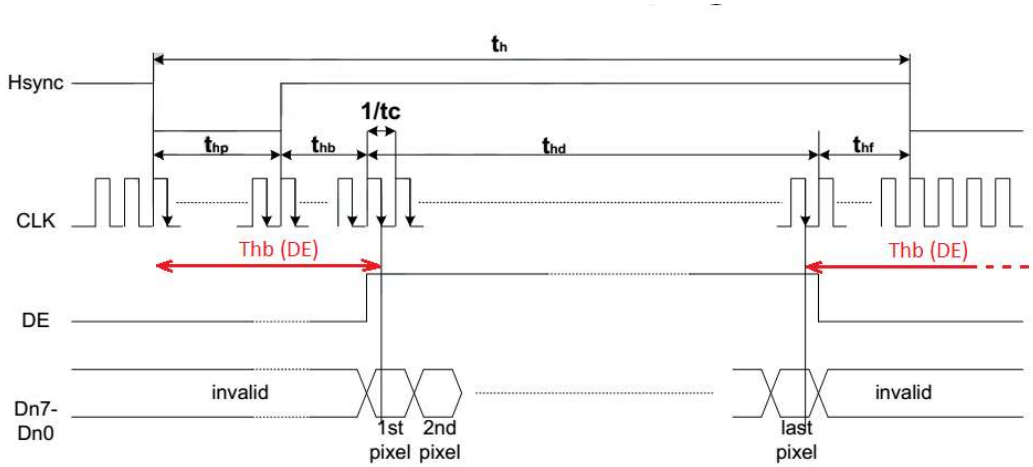


Ilustración 3.27: Relación entre el sincronismo horizontal y la habilitación de datos

De esta gráfica se observa que la temporización de borrado horizontal de la señal de habilitación de datos —Thb— es igual a la suma de del sincronismo horizontal más la temporización de sus dos pórticos —thp + thb + thf—. En el caso de la relación de la señal de habilitación de datos con el sincronismo vertical sucede algo similar:

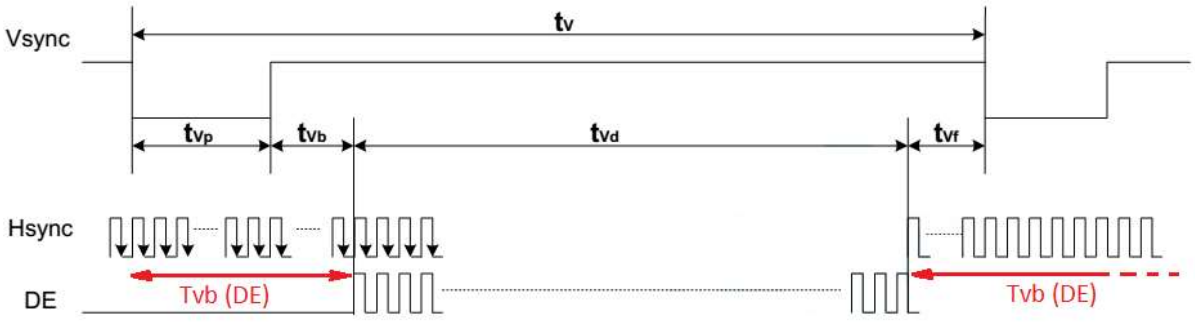


Ilustración 3.28: Relación entre el sincronismo vertical y la habilitación de datos

La temporización de borrado vertical de la señal de habilitación de datos, es igual a la suma del sincronismo vertical más sus dos pórticos — tvp + tvb + tvr—. Para conocer las posibles temporizaciones de sincronismos horizontal y vertical que correspondieran al display utilizado, se consulta las especificaciones de otro de 4,3

pulgadas del mismo fabricante y con la misma resolución, que dispone, además, de estas señales de sincronismo. Este display es el AT043TN24 de innoLux, y de sus especificaciones se deducen las siguientes relaciones:

Temporizaciones de señal «DE» display AT050TN33		Temporizaciones de señales de sincronismo VSYNC y HSYNC display AT043TN24		Hipotéticas temporizaciones de los sincronismo VSYNC y HSYNC display AT050TN33	
Borrado Vertical (Tvb)	16 líneas	Pulso Vertical (tvp)	10 líneas	Pulso Vertical	10 líneas
		Pórtico Vertical trasero (tvb)	2 líneas	Pórtico Vertical trasero	2 líneas
		Pórtico Vertical frontal (tvf)	2 líneas	Pórtico Vertical frontal	4 líneas
Borrado Horizontal (Thb)	45 puntos	Pulso Horizontal (thp)	41 puntos	Pulso Horizontal	41 puntos
		Pórtico Horizontal trasero (thb)	2 puntos	Pórtico Horizontal trasero	2 puntos
		Pórtico Horizontal frontal (thf)	2 puntos	Pórtico Horizontal frontal	2 puntos

Tabla 15: Inferencia de HSYNC y VSYNC del display utilizado —AT050TN33—

Para el caso de las temporizaciones hipotéticas del sincronismo vertical del display utilizado, los posibles valores de pulso vertical, pórtico trasero y delantero son los mostrados en las tres primeras filas de la columna sexta de la tabla. El tiempo de borrado vertical del display utilizado es de 16 líneas, y el pulso vertical y los dos pórticos verticales del display similar son de 10, 2 y 2, respectivamente. Estos tres últimos valores no suman las 16 líneas que son necesarias para el borrado vertical del display utilizado, así que alguno de los valores de las temporizaciones verticales del display similar, ha de ser mayor. El pulso vertical del display similar es de 10 líneas, un tiempo grande en comparación con sus pórticos, con lo que este tiempo será el utilizado para el display instalado. Quedan 6 líneas a repartir entre los dos pórticos. Se decide, de forma arbitraria, utilizar la misma temporización del pórtico vertical trasero del display AT043TN24 en el display instalado AT050TN33, con lo que se han de seleccionar, al menos 4 líneas para el hipotético pórtico delantero para el display instalado. Sin embargo, debidos a criterios de eficiencia, se aumenta el valor del pórtico delantero a 50, tal y como se aconseja por el fabricante de *TouchGFX* —ver sección 1.5—. Para el caso de las temporizaciones horizontales se procede de forma similar. En este caso el tiempo de borrado horizontal del display utilizado es de 45 puntos, y el pulso horizontal y sus dos pórticos son de 41, 2 y 2, respectivamente. Aquí no hay nada que discernir, ya que la suma de estos últimos valores coincide con el borrado horizontal del display utilizado. Estos valores son mostrados en las tres últimas filas de la columna sexta de la tabla anterior. Los valores de columna —sexta— serán los utilizados para configurar los parámetros de temporización del periférico LTDC, y son los valores de las constantes mostradas en el código anterior.

Esto se realiza mediante la siguiente porción de código, que va a continuación del mostrado anteriormente y que pertenece a la función *LCD_Config*:

```
1       /* Configuración de temporizaciones */
2       /* Ancho del sincronismo horizontal */
3       LTDC_InitStruct.LTDC_HorizontalSync = H_SYN_W-1;
4       /* Alto del sincronismo vertical */
5       LTDC_InitStruct.LTDC_VerticalSync = V_SYN_W-1;
6       /* Temporización acumulada con el pórtico horizontal trasero */
7       LTDC_InitStruct.LTDC_AccumulatedHBP = H_SYN_W+H_BACK_PORT-1;
8       /* Temporización acumulada con el pórtico vertical trasero */
9       LTDC_InitStruct.LTDC_AccumulatedVBP = V_SYN_W+V_BACK_PORT-1;
10      /* Configuración del ancho activo acumulado */
11      LTDC_InitStruct.LTDC_AccumulatedActiveW = H_SYN_W+H_BACK_PORT+WITH_ACTIVO-1;
12      /* Configuración del alto activo acumulado */
13      LTDC_InitStruct.LTDC_AccumulatedActiveH = V_SYN_W+V_BACK_PORT+HEIGHT_ACTIVO-1;
14      /* Configuración del ancho total */
15      LTDC_InitStruct.LTDC_TotalWidth = H_SYN_W+H_BACK_PORT+WITH_ACTIVO+H_FROM_PORT-1;
16      /* Configuración del alto total */
17      LTDC_InitStruct.LTDC_TotalHeigh = V_SYN_W+V_BACK_PORT+HEIGHT_ACTIVO+V_FROM_PORT-1;
18
19      /* Configuración de las componentes R,G,B el color de fondo del LCD */
20      LTDC_InitStruct.LTDC_BackgroundRedValue = 0;
21      LTDC_InitStruct.LTDC_BackgroundGreenValue = 0;
22      LTDC_InitStruct.LTDC_BackgroundBlueValue = 0;
23
24      LTDC_Init(&LTDC_InitStruct);
25
```

Código 3.28: Configuración de las temporizaciones del periférico LTDC

La estructura de configuración es *LTDC_InitStruct* de tipo *LTDC_InitTypeDef*, declarada al inicio de la función *LDC_Config*. Para la configuración de las temporizaciones, se utilizan los valores de las constantes y todos los parámetros estarán especificados decrementados en una unidad. Inicialmente se especificando el pulso de los sincronismos —horizontal en la línea 3, del código anterior, y vertical en la 5—. Luego, se van especificando las siguientes temporizaciones como valores acumulados con las temporizaciones anteriores. Así, para el caso de temporizaciones horizontales, se especifica la temporización acumulada del pulso horizontal con el pórtico trasero —línea 7—, la temporización acumulada con la dimensión horizontal activa —línea 11—, y la temporización horizontal total como la cantidad acumulada del pulso horizontal, los dos pórticos y la zona activa —línea 16—. Para el caso vertical, se tienen, la temporización acumulada hasta el pórtico trasero —línea 9—; la acumulada hasta la zona activa —línea 13— y la total —línea 17—. En la porción del código mostrado se indica, también, las componentes de color del fondo que tendrá la pantalla, que en este caso es negro —líneas de la 20 a la 22—. Este color de fondo se va a mezclar con las dos capas que puede gestionar el LTDC según la configuración que se especifique —se mostró un esquema del solapamiento de capas en la sección 3.5.1—. Toda esta configuración es llevada a cabo en la llamada a la función *LTDC_Init* que recibe como argumento la estructura rellena con los parámetros deseados para la inicialización.

Hasta aquí se configuran el barrido y color de fondo de la pantalla, pero el periférico LTDC manda la información hacia dos capas superpuestas sobre el color de fondo. Cada capa puede ser vista como una pantalla «lógica» superpuesta y puede tener información de transparencia en cada pixel —formato RGB8888— y/o tener información de transparencia de toda la capa.

No es obligatorio que la capa utilice toda la superficie de la pantalla, sino que pude mostrar solo parte del buffer de pantalla en una zona rectangular de la misma [8]:

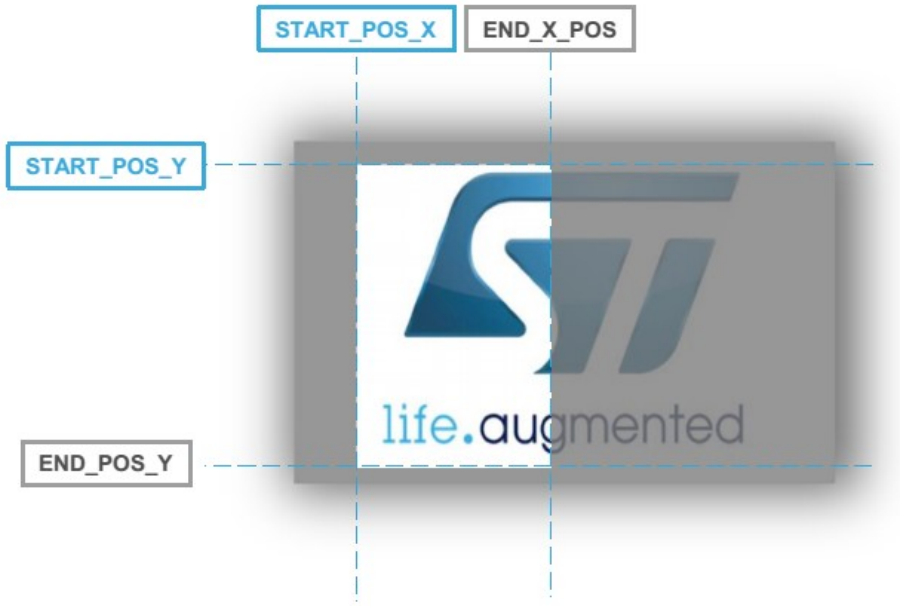


Ilustración 3.29: Parámetros dimensionales de las capas del periférico LTDC

Los parámetros dimensionales de la capa son los mostrados en la figura, consistentes en especificar la coordenada superior izquierda de la capa —en puntos de pantalla— y el ancho y alto de la misma —igualmente en puntos—. La El código referente a lo comentado es el siguiente:

```
1   /* Configuración de la capa1 del LTDC ------------------------------------------*/
2
3   /* Configuración de ventana para la capa 1 */
4   /* La capa 1 va ocupara toda la dimensión de la pantalla  */
5   LTDC_Layer_InitStruct.LTDC_HorizontalStart = H_SYN_W+H_BACK_PORT;
6   LTDC_Layer_InitStruct.LTDC_HorizontalStop = H_SYN_W+H_BACK_PORT+WITH_ACTIVO-1
7   LTDC_Layer_InitStruct.LTDC_VerticalStart = V_SYN_W+V_BACK_PORT;
8   LTDC_Layer_InitStruct.LTDC_VerticalStop = V_SYN_W+V_BACK_PORT+HEIGHT_ACTIVO-1;
9
10  /* Formato de pixel para la capa 1*/
11  LTDC_Layer_InitStruct.LTDC_PixelFormat = LTDC_Pixelformat_RGB565;
12
13  /* Configuración del valor Alpha para la capa 1 (255 totalmente opaca) */
14  LTDC_Layer_InitStruct.LTDC_ConstantAlpha = 255;
15
16  /* Configuración de factores de mezclado de la capa 1 */
17  LTDC_Layer_InitStruct.LTDC_BlendingFactor_1 = LTDC_BlendingFactor1_PAxCA;
18  LTDC_Layer_InitStruct.LTDC_BlendingFactor_2 = LTDC_BlendingFactor2_PAxCA;
19
20  /* Color por defecto para la capa 1 */
21  LTDC_Layer_InitStruct.LTDC_DefaultColorBlue = 0;
22  LTDC_Layer_InitStruct.LTDC_DefaultColorGreen = 0;
23  LTDC_Layer_InitStruct.LTDC_DefaultColorRed = 0;
24  LTDC_Layer_InitStruct.LTDC_DefaultColorAlpha = 0;
25
26  /* Dirección de inicio donde se localiza la capa 1 */
27  LTDC_Layer_InitStruct.LTDC_CFBStartAdress = frameBuf0;
28
```

Código 3.29: Configuración de la capa1 del LTDC

Para el uso de la librería TouchGFX solo es necesaria la capa 1 del LTDC, y además esta capa ha de ocupar toda la pantalla. Esto se especifica con las coordenadas de inicio de la capa localizadas al inicio del área activa de la pantalla —líneas 5 y 7 (el inicio de la capa está después de los pulsos de sincronismo y los pórticos traseros)— y

sus dimensiones de ancho y alto a los de la pantalla —líneas 6 y 8 —. El formato de color de la capa —el bloque PFC de la capa, ver Ilustración 3.20 de la sección 3.5.1— será RGB565 —línea 11—, que es el soportado por la versión utilizada de TouchGFX. Ya que el periférico LTDC dispone de un fondo de color y dos capas, la mezcla entre ellas —blend—, está siempre presente [8]:

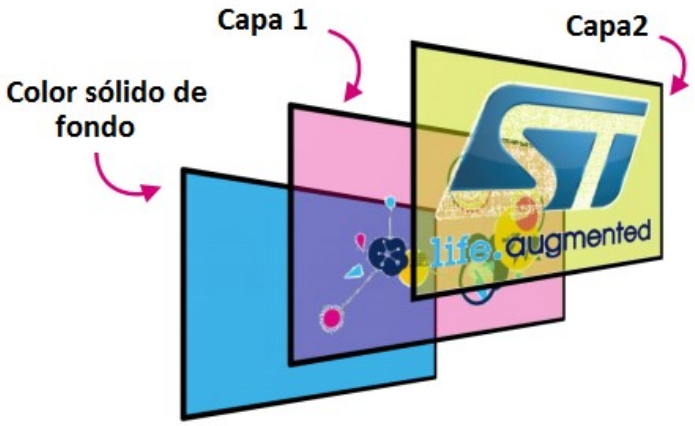


Ilustración 3.30: Mezcla de las capas en el periférico LTDC

Esto es independiente si alguna capa está activa o no. En el caso de este proyecto, solo se utiliza la capa 1 y el fondo de color —que como ya ha sido expuesto es de color negro—, con lo la mezcla solo será de ambos. La mezcla se realizará teniendo en cuenta la información de transparencia de cada pixel junto con la general de la capa. Para el caso de la capa1, la información suministrada de transparencia general de la capa hace que sea totalmente opaca —línea 14 del código anterior—. Para la mezcla entre capas —y fondo de color— se utiliza la siguiente expresión [4]:

$$BC = BF1 \times C + BF2 \times C_S$$

Donde *BC*, es la componente del color del pixel resultante de la mezcla; *BF1*, es el factor de mezcla 1, que multiplica a la componente de color del pixel de la capa actual; *C*, es la componente de color del pixel de la capa actual; *BF2*, es el factor de mezcla 2, que multiplica a la componente de color del pixel de la mezcla de capas subyacentes, y C_S, es la componente de color de la mezcla de capas subyacentes. El factor de multiplicación *BF1* puede ser la constante alpha de la capa en cuestión —en este caso el de la capa 1— entre 255, o la multiplicación de la constante alpha de la capa en cuestión, por la información alpha del punto, cada uno de ellos entre 255. En el caso de *BF2*, puede ser uno menos la constante alpha de la capa en cuestión —en este caso, capa 1— entre 255, o uno menos la multiplicación de la constante alpha de la capa en cuestión por la información alpha de punto, cada elemento de la multiplicación dividido entre 255. Todas estas operaciones son «transparentes» para el programador, lo único que hay que suministrar es la constante alpha de toda la capa y especificar si la mezcla con la capas de abajo va a ser teniendo solo en cuenta la información de transparencia de toda la capa o la de toda la capa con la información de cada punto. En este caso se ha suministrado información de transparencia opaca para toda la capa 1 y la mezcla será teniendo en cuenta la información de transparencia de toda la capa y la de capa pixel —líneas 17 y 18 donde se especifica que los factores *BF1* y *BF2* serán referentes a la información de transparencia de toda la capa y la de cada pixel, constante LTDC_BlendingFactorX_PAxCA —.

Así, como se ha suministrado la constante de transparencia para toda la capa 1 a 255 —opacidad total—, la expresión anterior queda:

$$BC = \frac{PA_{C1}}{255} \times \frac{CA_{C1}}{255} \times C1 + \left(1 - \frac{PA_{CF}}{255} \times \frac{CA_{C1}}{255}\right) \times C_F$$

$$= \frac{PA_{C1} \times 255}{255^2} \times C1 + \left(1 - \frac{PA_{CF} \times 255}{255^2}\right) \times C_F$$

$$= \frac{PA_{C1}}{255} \times C1 + \left(1 - \frac{PA_{CF}}{255}\right) \times C_F \rightarrow \ como\ PA_{CF}\ es\ 255\ y\ \ C_F\ es\ cero\ (color\ negro) \rightarrow$$

$$BC = \frac{PA_{C1}}{255} \times C1$$

Donde *BC* es la componente de color de cada punto de la mezcla entre la capa1 y el fondo, PA_{C1} es la información de transparencia de cada punto de la capa 1, CA_{C1} es la constante de transparencia de toda la capa 1, *C1* es la componente de color de cada punto de la capa 1, PA_{CF} es la información de transparencia de cada punto del fondo y *CF* es la componente de color de cada punto del fondo. Como la capa 1 es totalmente opaca, el resultado de la mezcla es que solo se tiene en cuenta las componentes de color de la capa1.

A parte de esta información, se debe dar el color que debe tener la capa en caso que no ocupe toda la pantalla —no es el caso--. Esta información del color de «relleno» así como su transparencia es suministrada entre las líneas 21 a 23 del código anterior —color negro y totalmente transparente—.

La información que se ha de presentar en pantalla está contenida en la memoria SDRAM externa donde se albergan los dos búferes de pantalla. Para la capa 1, esta dirección de memoria está especificada por la variable frameBuf0 —línea 27 del código anterior—, declarada como global en el archivo BoardConfiguration.cpp —global de alcance fichero—. El contenido de esta variable es la dirección de inicio donde está mapeada la SDRAM dentro del esquema de direccionamiento de micro. *TouchGFX* cambiará el valor de la dirección de memoria de la capa 1, donde esta obtiene sus datos, al producirse el intercambio de búferes —esto es una suposición ya que no se tiene acceso a esta parte del código—.

Para finalizar con la configuración de la capa 1, se ha de especificar cómo se muestra la información obtenida del buffer de memoria:

```
1     LTDC_Layer_InitStruct.LTDC_CFBLineLength = ((WITH_ACTIVO*2)+3);
2
3     LTDC_Layer_InitStruct.LTDC_CFBPitch = (WITH_ACTIVO*2);
4
5     /* Configuración del número de líneas */
6     LTDC_Layer_InitStruct.LTDC_CFBLineNumber = HEIGHT_ACTIVO;
7
8     /* actualización de los parámetros de configuración en la capa 1 */
9     LTDC_LayerInit(LTDC_Layer1, &LTDC_Layer_InitStruct);
10
11    /* Fin de configuración de la capa 1 ------------------------*/
12
13    /* Habilitación del difuminado a la salida */
14    LTDC_DitherCmd(ENABLE);
15
16
```

Código 3.30: Configuración para la obtención de información en la capa 1 del periférico LTDC

En la línea 1 de la porción de código, se especifica cuál es el ancho de una línea, mostrada en la capa1, expresada en bytes —la configuración del periférico necesita añadirle la cantidad de 3 a este valor—. También es necesario configurar el «Pitch», que se podría traducir como el paso, en bytes entre el inicio de una línea y el comienzo de la siguiente, es decir, a la hora de mostrar una línea en pantalla, saber cuál es la dirección de memoria de inicio de la siguiente. Para acabar con la configuración referente a la extracción de información del búfer de memoria, se debe especificar cuantas «líneas de memoria» constituyen la representación de la capa —línea 7—. Gráficamente, estos conceptos se resumen en la siguiente figura [8]:

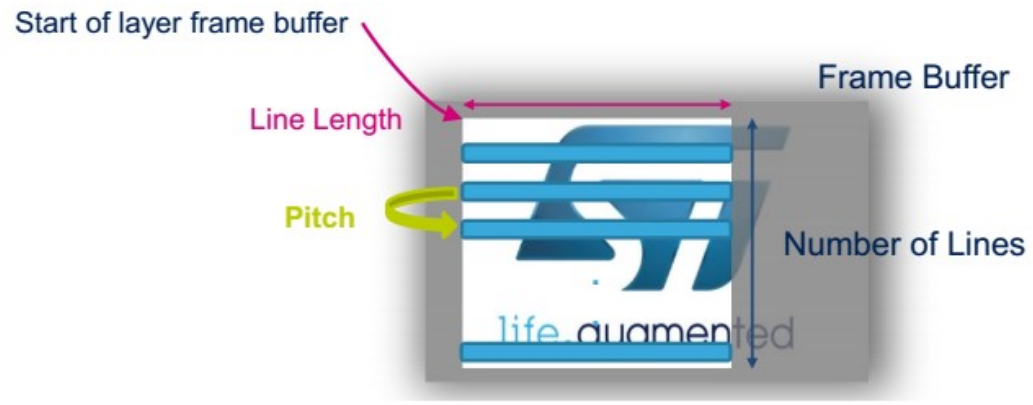


Ilustración 3.31: Parámetros del buffer de memoria para configurar la capa 1 del periférico LTDC

En esta figura se muestra la parte de la imagen a mostrar en la capa 1, pero no como representación en pantalla, sino como representación en memoria. Se debe conocer la dirección de inicio del buffer —ya comentada con anterioridad—, que es el punto de inicio de la imagen que se quiere mostrar en la capa. Para poder representar una línea en la capa, es necesario conocer qué cantidad de memoria hay que extraer; este es el parámetro especificado en la línea 1 del código anterior y que corresponde a la longitud de la línea en puntos de la capa multiplicado por cuanto ocupa cada punto en memoria. Para poder representar la siguiente y sucesivas líneas en la capa, una vez representada la primera, se ha de saber cuánto se tiene que saltar en memoria desde el inicio de una línea, que está especificado en la línea 3 del código. Esto puede ser trivial cuando la capa ocupa la totalidad de la pantalla —como es el caso—, donde la dirección de memoria de inicio de la siguiente línea a representar en la capa, es la dirección del siguiente dato en memoria. Pero si la capa no ocupa toda la pantalla sino que ocupa solo una región de la misma, es necesario saber cuántas direcciones se han de saltar desde la dirección del último dato representado en la última línea, hasta alcanzar la siguiente —Pitch—. Lo último que se ha de conocer es el número de líneas representadas en la capa —línea 7 del código— para conocer cuántos saltos de memoria —Pitch— hay que hacer para representar toda la capa y volver a la dirección de inicio para iniciar un nuevo cuadro.

Finalmente, todos los valores de configuración de la capa son tenidos en cuenta en la línea 10 del código anterior, mediante la llamada a la función *LTDC_LayerInit*, que toma como segundo argumento la estructura *LTDC_Layer_InitStruct* —declarada en el inicio de la función *LCD_Config*—, de tipo *LTDC_Layer_InitTypeDef*, que ha ido siendo rellenada a lo largo de la configuración de la capa, y como primer argumento toma la identificación de la capa que se quiere configurar, *LTDC_Layer1* —la capa 1—.

Para evitar imágenes con degradados escalonados, ya que el formato es RGB565, se habilita la función de difuminado del LTDC que afecta a todas las capas —línea 16 del código—.

3.5.1.5 Descripción del funcionamiento de la pantalla táctil de tipo resistivo.

La pantalla táctil que viene instalada con el display seleccionado — InnoLux AT050TN33— es de tipo resistivo de cuatro hilos y su estructura básica es la siguiente [29]:

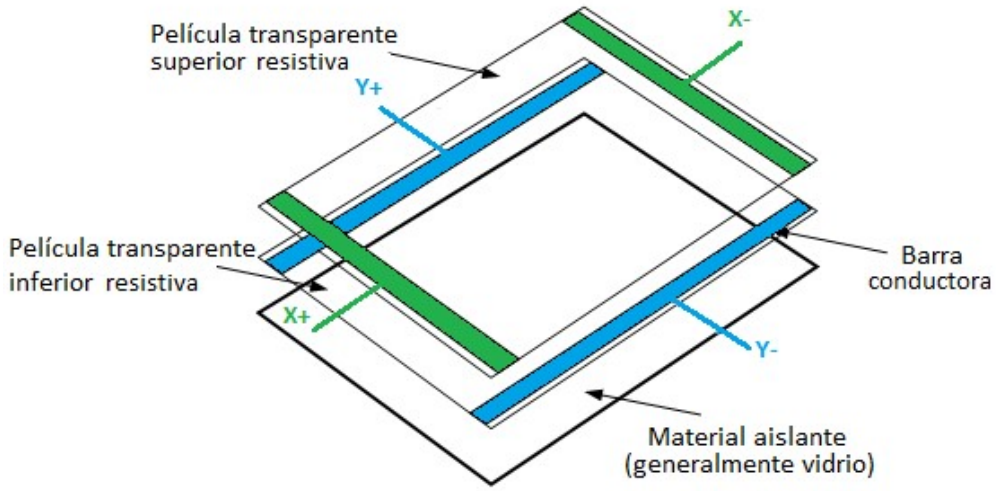


Ilustración 3.32: Estructura de una pantalla táctil resistiva de cuatro hilos

Consta de dos películas conductoras transparentes, una colocada encima de otra y separadas por una capa espaciadora transparente, no mostrada en la figura, que las mantienen aisladas si no se produce una pulsación sobre la pantalla. Las películas conductoras están fabricadas de tal modo que su resistencia varia de forma lineal solo en un eje. Así, en la película superior, su resistencia varía solo en el eje horizontal, mientras que para una misma vertical su resistencia permanece constante —idealmente—. De similar modo, en la película inferior la resistencia varía solo en el eje vertical mientras que para una misma horizontal su resistencia permanece constante. Las dos películas descansan sobre un material aislante que generalmente es el vidrio del display sobre el que se instala, pero esto no tiene que ser siempre así, se puede utilizar una pantalla táctil sin necesidad de tener un display debajo como pueda ser el caso de un dispositivo que recoja nuestra firma. Cada película tiene en sus extremos unas barras conductoras, perpendiculares al eje en el que varía su resistencia, donde se insertan los terminales. Estos terminales van a ser utilizados tanto para suministrar una tensión de medida como para realizar la medida en sí. Por ello, uno de los terminales de cada película se marca con el signo positivo —terminal que recibirá la tensión de medida— y el otro con el negativo —el que irá conectado a masa cuando se suministre tensión a la película—. Los nombres de los terminales de una misma película se denominan con el eje que dicha película controla, diferenciando cada terminal de una misma película por su signo.

Cuando se produce una pulsación sobre la pantalla, las dos películas entran en contacto, pero solo en la zona de la pulsación —asegurado por la capa espaciadora que separa a ambas películas—. Para conocer las coordenadas de la pulsación de la pantalla, se sigue un proceso de dos pasos, tal y como se muestra en la siguiente figura [31]:

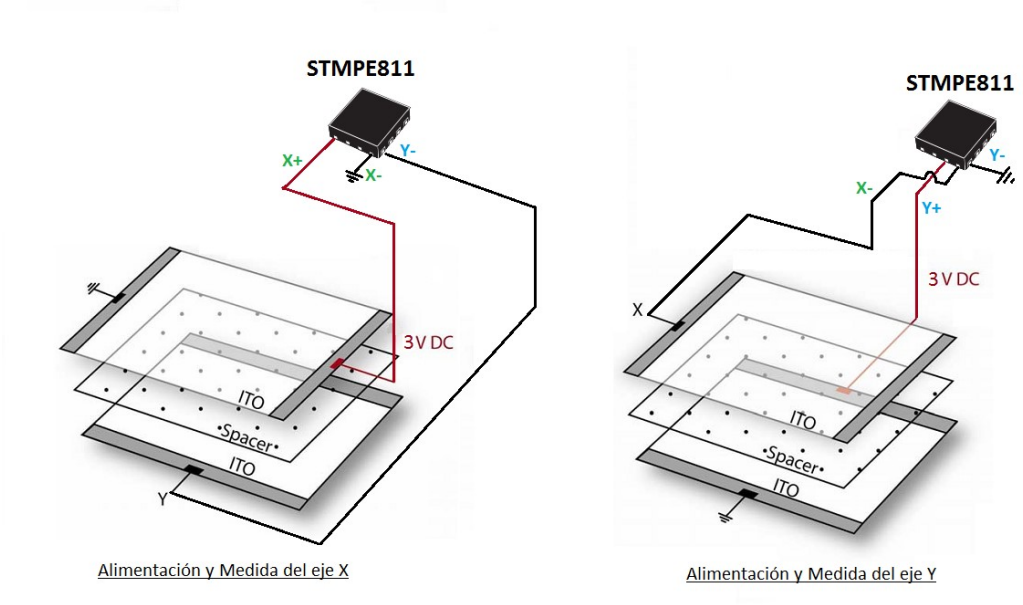


Ilustración 3.33: Pasos de medida en una pantalla resistiva de cuatro hilos

Se comienza sometiendo a tensión una de los dos películas, entre sus terminales positivo y negativo. En la figura, primeramente, se muestra la alimentación de la película superior —en este caso la película que controla el eje horizontal (parte izquierda de la figura) — y se realiza la medida en uno de los terminales de la otra película —la inferior—. Luego, se alimenta la otra capa —la inferior—, entre sus dos terminales, y la medida se realiza por uno de los terminales de la capa que no ha sido alimentada —parte derecha de la figura—. Todo este proceso es llevado a cabo mediante un circuito controlador de pantalla táctil resistiva, que en el caso de este proyecto es el STMPE811 y que será descrito en la siguiente sección. Para comprender mejor el proceso de medida, se muestra a continuación la representación esquemática de la pantalla táctil en sus dos pasos de medida [32]:

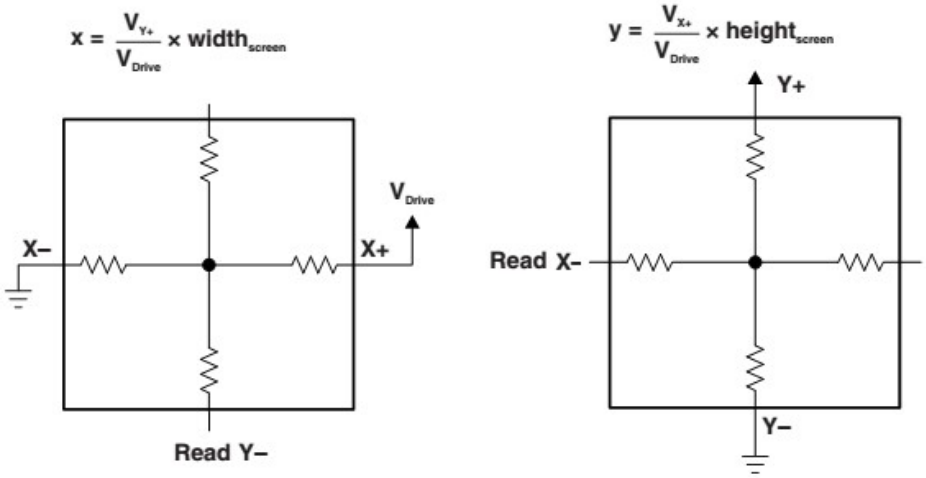


Ilustración 3.34: Esquema de la pantalla táctil en el proceso de medida

En el primer paso de medida, al alimentar la película «X», se crea un divisor de tensión resistivo debido al contacto de las dos películas por la pulsación realizada. Para tener una medida de posición horizontal, se ha de adquirir la tensión en este divisor —tensión entre el punto de contacto y el terminal «X-» —, cosa que se hace desde un terminal de la otra película, que en el caso de la figura es desde el terminal «Y-». Sin embargo, hay una cierta resistencia entre este último terminal y el punto de contacto, donde se quiere medir la tensión debida exclusivamente al divisor resistivo horizontal. Esto no tiene porque ser un inconveniente ya que la impedancia interna del convertidor analógico-digital —ADC—es mucho mayor que la resistencia total de cada una de las películas. Al finalizar este paso de medida, se obtiene una tensión que es proporcional a la posición horizontal de la pulsación. Tal y como se muestra en la fórmula que aparece en la parte superior de la representación de la pantalla táctil en la zona izquierda de la figura, la posición horizontal es obtenida mediante la división de la tensión medida respecto de la utilizada en la alimentación de la película, multiplicada por la dimensión horizontal total de pantalla. El segundo paso de medida es prácticamente igual al primero con la diferencia que cambia la coordenada del divisor resistivo de tensión a medir. La forma en que se pasa de tensión a coordenada longitudinal es para un caso ideal. Más adelante, al confeccionar el driver de la pantalla táctil para *TouchGFX*, se mostrará un estudio de regresión para ambas coordenadas.

3.5.1.6 Control táctil de la pantalla: descripción del periférico stmpe811 de la placa 32F429IDiscovery.

Este periférico es un integrado que posibilita «expandir» las capacidades de comunicación del micro que lo utiliza. Permite ampliar hasta 8 canales de entrada/salida de propósito general, controlar una pantalla táctil resistiva de 4 hilos o adquirir señales analógicas mediante un ADC de 12 bits a través de 8 entradas seleccionables. La placa STM32F429I-DISCO —usada en este proyecto— lo utiliza para controlar la funcionalidad táctil del display de 2,4 pulgadas que lleva integrado. Este proyecto se desarrolla a partir de esta placa, pero, al considerar insuficiente el tamaño de la pantalla instalada, se opta por acoplar un display mayor —de 5 pulgadas—. Sin embargo se pretende reutilizar este circuito para darle la funcionalidad táctil a la mueva pantalla. Por ello es necesario conocerlo, básicamente en el aspecto hardware y sobre todo, en el software que la placa ya utiliza. Desde el punto de vista hardware, se ha de conocer su esquema simple de bloques y patillaje, sobre todo el del encapsulado, ya que será prácticamente desde el mismo, donde cablear hacia la nueva pantalla. Desde el punto de vista software, se reutilizará parte del código, especialmente el de configuración del periférico

El esquema general de este circuito es el siguiente [28]:

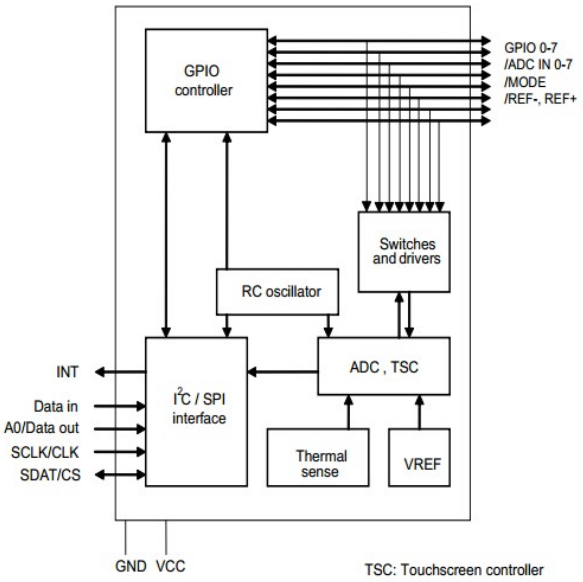


Ilustración 3.35: Esquema del periférico externo STMP811 de la palca STM32F429I-DISCO

Consta básicamente del controlador de entrada/salida —GPIO—, que permite expandir las entradas y salidas digitales del micro hasta un total de ocho, o darles las funciones alternativas de entradas para la adquisición analógica o control de pantalla táctil. Por ello, internamente dispones de un bloque de conversión analógico/digital —ADC— con funcionalidad para gestionar una pantalla táctil —TSC— y un puerto de comunicaciones serie —I2C/SPI— por donde controlar la configuración o mandar y leer datos.

Dispone de otros elementos accesorios como son el oscilador interno, un sensor de temperatura, una referencia de tensión, un buffer de 128 datos de 32 bits y un banco de registros de configuración, de estado y de datos —no mostrado en la figura anterior—.

La distribución de pines se muestra en la siguiente tabla [28]:

Pin	Name	Function
1	Y-	Y-/GPIO-7
2	INT	Interrupt output
3	A0/Data Out	I^2C address in Reset, Data out in SPI mode
4	SCLK	I^2C/SPI clock
5	SDAT	I^2C data/SPI CS
6	V_{CC}	1.8 –3.3 V supply voltage
7	Data in	SPI Data In
8	IN0	IN0/GPIO-0
9	IN1	IN1/GPIO-1/MODE In RESET state, MODE selects the type of serial interface "0" - I^2C "1" - SPI
10	GND	Ground
11	IN2	IN2/GPIO-2
12	IN3	IN3/GPIO-3
13	X+	X+/GPIO-4
14	Vio	Supply for touchscreen driver and GPIO
15	Y+	Y+/GPIO-5
16	X-	X-/GPIO-6

Tabla 16: distribución de pines del integrado STMPE811

Dispone en total de 16 pines, alguno de ellos con varias funciones, pero las que interesan son las dedicadas a la pantalla táctil, es decir, Y+ e Y-, para el control vertical y X+ y X- para el horizontal. También son necesarias las señales de comunicación serie con el micro —pines 4 y 5—, en configuración I2C —en contraposición a SPI, que también está disponible—, ya que, ese es el modo utilizado para la comunicación con la pantalla integrada. Además, algunos de los pines tienen sus niveles fijados, estableciendo una configuración para el integrado que no puede ser cambiada. Esto se observa en la parte del esquema de la placa *Discovery* donde se muestra este periférico [6]:

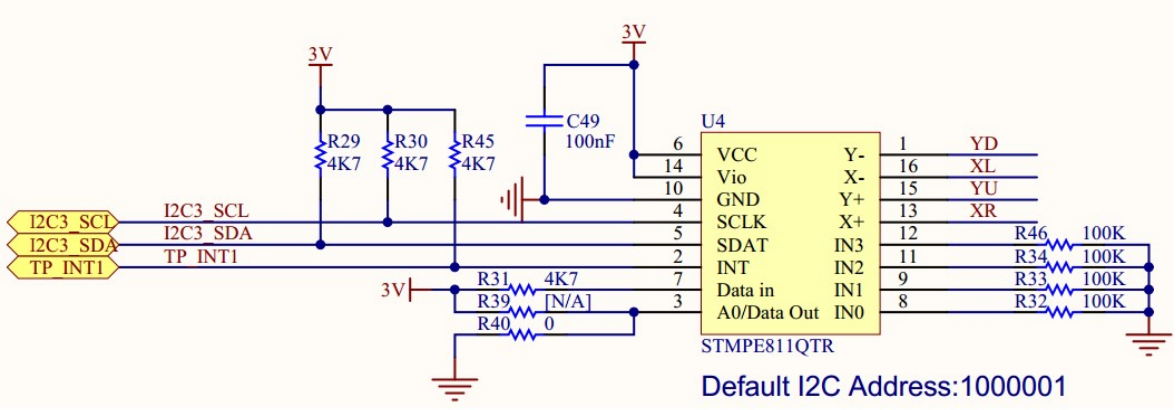


Ilustración 3.36: Conexiones del integrado STMP811 en la placa 32F429IDiscovery

El pin 9 permite seleccionar entre I2C —nivel cero— o SPI —nivel uno—, pero en la placa está fijado a cero a través de la resistencia R33. La dirección del I2C puede ser configurada mediante el pin 3 entre los valores 0x82 —si este pin está a nivel uno— o a 0x88 —si este pin está a nivel cero—. El pin queda fijado a nivel cero a través del puente R40, con lo que su dirección será 0x82. Este mismo pin —pin 3— funciona como señal de datos de salida SPI —MISO— si el pin 9 selecciona este modo de comunicación —pero se acaba de describir que el modo de comunicación seleccionado es I2C—. La función del pin 7 es exclusivamente para entrada de datos —MOSI—, pero como la comunicación SPI no es utilizada, se conecta directamente a 3V a través de R31. Los pines 4 y 5 constituyen las líneas de comunicación I2C —reloj y datos, respectivamente— que se conectarán con los pines del micro que actuarán como líneas del periférico interno I2C3 en modo máster —que, entre otras cosas, será

configurado por el software utilizado por la palca en relación con este integrado—. Estas líneas necesitan sus respectivas resistencias de «pull-up» constituidas por R29 my R30. El pin 2 es utilizado para indicar al micro la solicitud de atención de interrupción, disparándose esta señal cuando se realiza un toque sobre la pantalla —va a depender de la configuración del integrado—. Los pines de alimentación son el 6 y el 10 —VCC y GND respectivamente— y la tensión utilizada para medir la resistencia horizontal y vertical de la pantalla táctil —para detectar la coordenada pulsada— es cogida de la alimentación —pin 14 Vio—. Los pines 8, 9, 11 y 12 son las entradas 0, 1, 2 y 3 al convertidor ADC, que al no ser utilizadas son conectadas a GND a través de resistencias de 100K. Los pines 13, 15, 16 y 1, son las entradas 4, 5, 6 y 7 del ADC o las entradas X+, Y+, X- e Y- para testear la pantalla táctil —que será la configuración que realice el software de inicialización del periférico utilizado por la placa—. Estas señales son etiquetadas, respectivamente, dentro del esquema de la placa, como XR —derecha—, YU —arriba—, XL —izquierda—, YD —abajo—. El problema físico que presentan estas señales, es la dificultad en la accesibilidad. Las otras señales que gobiernan la presentación de imagen en pantalla, son accesibles mediante pines en los conectores P1 y P2 de la placa *Discovery*. Sin embargo, la tira de pines del circuito impreso donde está soldado el display de 2,4 pulgadas, integrado con la placa —donde, además de las señales para mandar la información de la imagen al display, están las cuatro señales de control de la pantalla táctil—, no aparece en los esquemas de la documentación de la palca como un conector, imposibilitando la identificación de las señales. Afortunadamente, esta información puede ser extraída del fabricante del display. El display es el modelo SF-TC240T-9370-T del fabricante SAEF Technology y del datasheet se observa la siguiente información [26]:

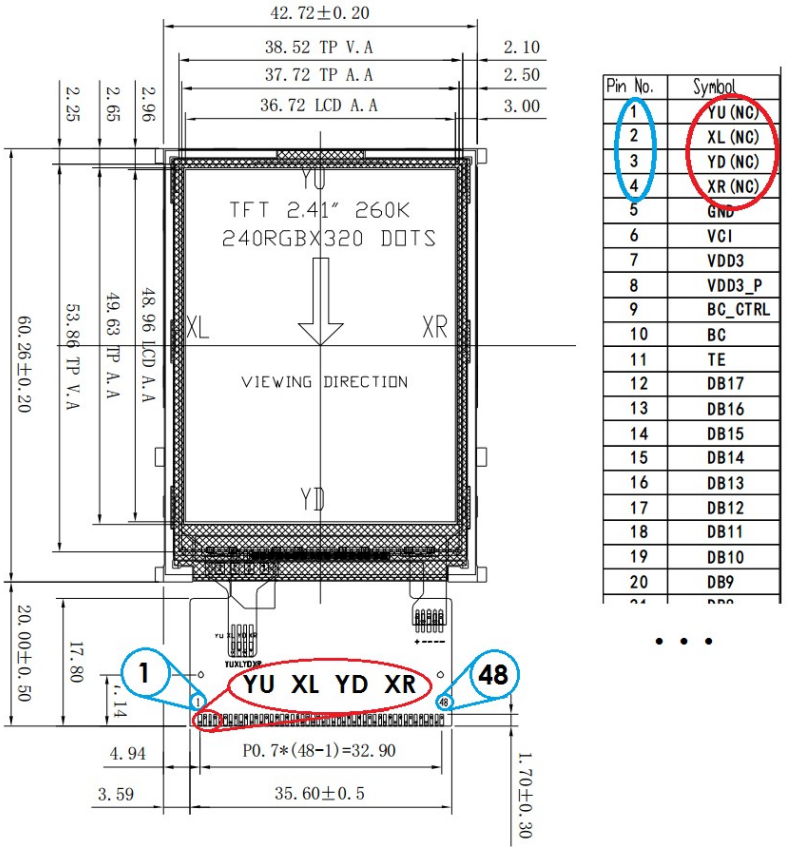


Ilustración 3.37: Display de 2,4 pulgadas integrado en la placa STM32F429I-DISCO

Tal y como se muestra en la figura, con el display bocarriba y el cable plano desplegado, las señales de control de la pantalla táctil, YU, XL, YD y XR, quedan en este orden, en la zona izquierda de la tira de pines del cable plano. Sabiendo esta distribución, se pueden identificar estas señales en la tira de pines de la placa *Discovery* donde va soldado el display [6]:

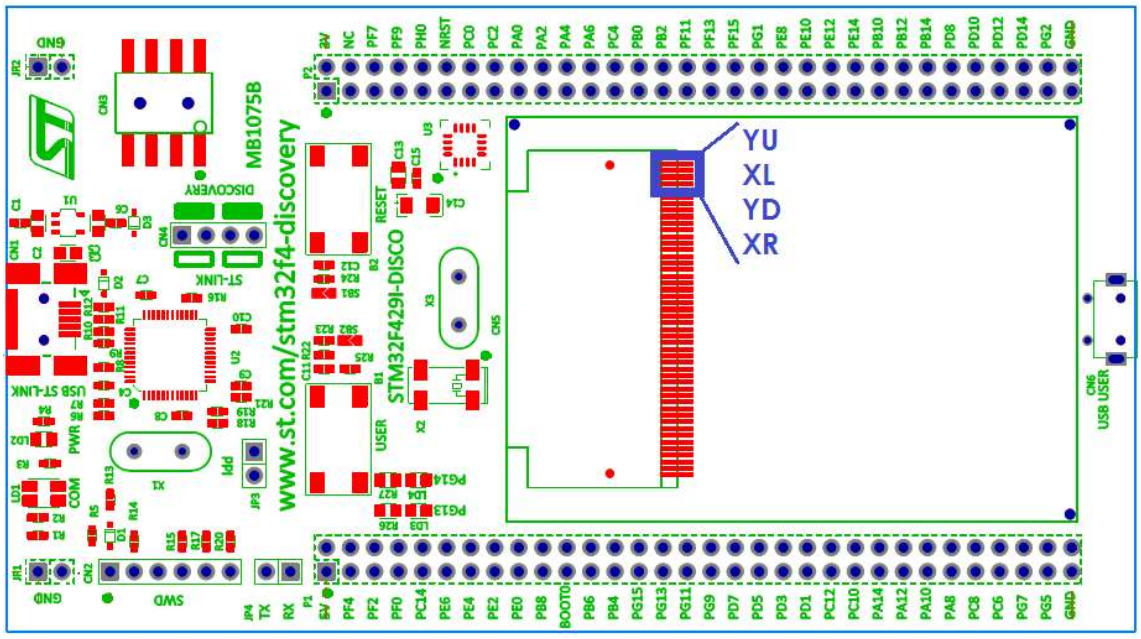


Ilustración 3.38: Señales para el control táctil de la placa STM32F429I-DISCO

Una vez identificadas estas señales, se trata de acceder a ellas de forma externa a la placa. Para ello, se crea un ingenio en la zona del circuito en el que se situaba anteriormente el display original, realizado con placas de circuito impreso para prototipos, donde se instala un conector con cuatro pines, con las señales de la pantalla táctil ruteadas como se muestra en la siguiente figura:

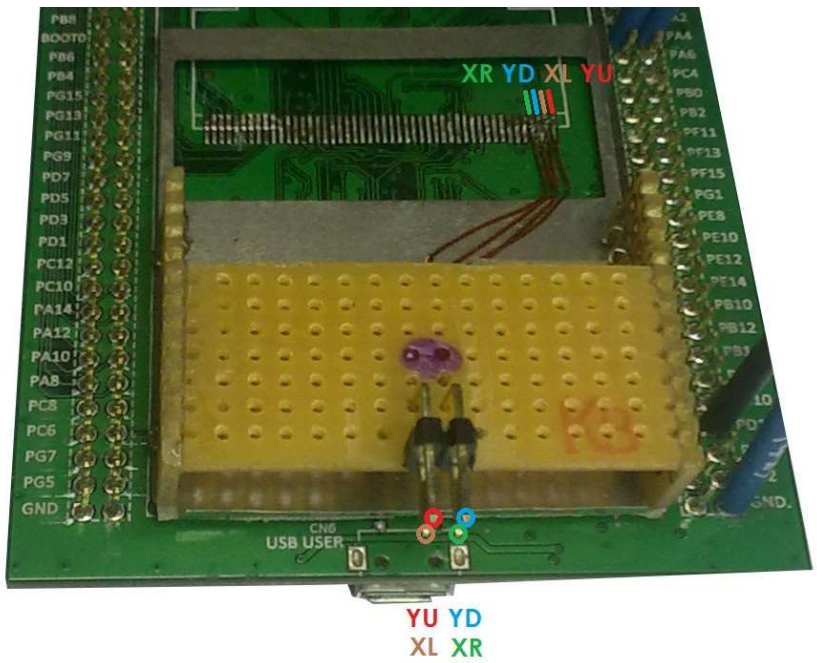


Ilustración 3.39: Instalación del conector del control táctil en la placa Discovery

Estas señales irán conectadas al terminal de la placa de prototipos base encargado de recoger las conexiones de la pantalla táctil mediante un cable plano. El detalle del conector en la placa base es el siguiente:

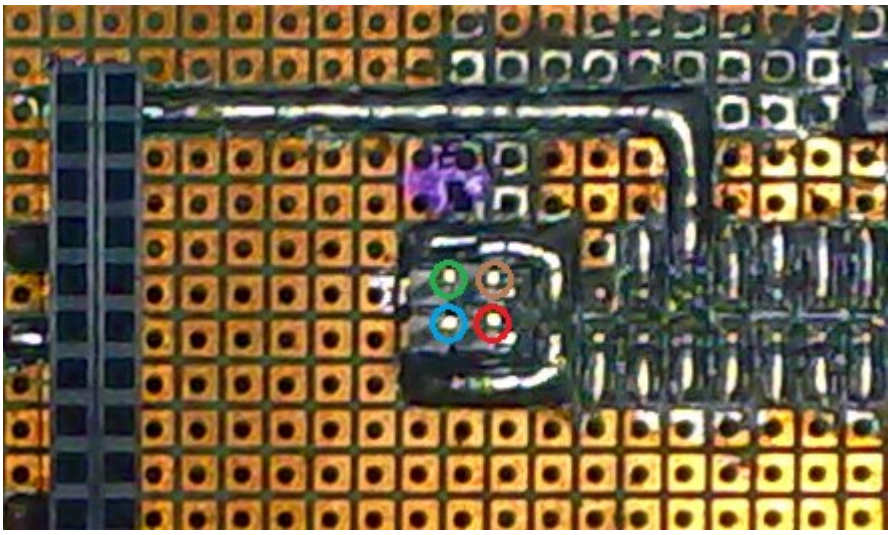


Ilustración 3.40: Conector de la señales de la pantalla táctil en la placa base

En principio no es crítico intercambiar las señales entre sí, mientras sigan perteneciendo a la misma coordenada táctil —por ejemplo, se pueden intercambiar la señales YU e YD, pero no las señales YU y XD—. Esta permisividad es debido a que no se utilizará directamente la lectura que ADC del integrado STMPE811 entrega al ser tocada una zona de la pantalla, sino que será pasada por una función que la convierta desde las cuentas del ADC a puntos de pantalla —estas funciones serán mostradas en la siguiente sección 3.5.1.7—.

Las funciones de la librería de la placa *Discovery*, referentes al integrado STMPE811, que serán reutilizadas para crear el driver táctil necesario en la librería *TouchGFX*, son las siguientes:

```
1  // Función de inicialización
2  IOE_Config();
3
4  // Función de lectura de registro
5  I2C_ReadDeviceRegister(IOE_REG_TP_CTRL)
6
7  // Funciones de lectura del buffer
8  I2C_ReadDataBuffer(IOE_REG_TP_DATA_X);
9  I2C_ReadDataBuffer(IOE_REG_TP_DATA_Y);
10
11 // Funciones de escritura de registro
12 I2C_WriteDeviceRegister(IOE_REG_FIFO_STA, 0x01);
13 I2C_WriteDeviceRegister(IOE_REG_FIFO_STA, 0x00);
```

Código 3.31: Funciones de la librería de la placa Discovery referentes al integrado STMP811

Para utilizar estas funciones es preciso incluir el archivo de cabecera *stm32f429i_discovery_ioe.h* en todos aquellos archivos fuente donde sean llamadas. No se va a entrar en describir profundamente lo que hacen y cómo lo hacen, solo qué hacen de forma somera, ya que no es el objetivo de este proyecto y el código fuente está disponible para su revisión y/o modificación.

La primera función mostrada —línea 1—, *IOE_Config*, configura tanto el micro STM32F429 como el periférico externo constituido por el integrado STMPE811, para que puedan comunicarse y que este último examine las pulsaciones de la pantalla. Cuando configura el micro, hace que su periférico interno I2C3, esté presente en el exterior a través del pin 8 del puerto A, con la señal del reloj del I2C SCL, y del pin 9 del puerto C como señal de datos —SDA— del bus I2C; pines que están conectados a las señales homónimas del integrado STMPE811 a través de pistas en la placa *Discovery*. Al configurar el circuito extensor de entradas/salidas, se modifica el contenido varios registros de configuración para que este pueda capturar los toques sobre pantalla. En definitiva, esta función es necesaria llamarla inicialmente para que se pueda usar la pantalla táctil. La función

de la línea 5 — *I2C_ReadDeviceRegister*—, permite leer el contenido de un registro —el especificado en el argumento de la función— del integrado STMPE811. El registro que se especificará en la llamada a esta función es el que se encuentra en la dirección 0x40, que corresponde a la constante IOE_REG_TP_CTRL y es el encargado de controlar el sistema táctil, permitiendo la configuración —realizada a través de *IOE_Config*— y estado del mismo —si se ha tocado o no la pantalla—. Las dos siguientes funciones —líneas 8 y 9—, son realmente la misma con diferente argumento de entrada. Sirven para leer el contenido del buffer interno del STMPE811 por el bus I2C para obtener la coordenada «x» y la coordenada «y» de la zona de la pantalla pulsada —mediante las constantes respectivas IOE_REG_TP_DATA_X e IOE_REG_TP_DATA_Y, que corresponden a las direcciones de los registros 0x4D y 0x4F—. La diferencia fundamental entre las funciones *I2C_ReadDeviceRegister* e *I2C_ReadDataBuffer*, es que la primera se usa para leer registros de 8 bits mientras que la segunda se usa para registros de 16 bits. Finalmente se usa la función *I2C_WriteDeviceRegister* que sirve para escribir un dato en un registro interno del integrado STMPE811. En concreto, se utiliza el registro localizado en la dirección 0x4B —IOE_REG_FIFO_STA— para mandar la orden de reinicio del buffer de datos táctiles —línea 12—, una vez leídos los datos y la orden de liberación del reinicio —línea 13— para permitir una nueva lectura.

3.5.1.7 Driver táctil del display.

Ya que se ha procedido a instalar una nueva pantalla, con características táctiles diferentes a la original, es necesario informar a la capa HAL de TouchGFX —capa específica para la placa STM32F429-DISCO— de dicho cambio. Para ello, TouchGFX dispone de varios mecanismos, el más versátil es crear el driver táctil desde cero a través de la clase abstracta *TouchController*, definida en el archivo de cabecera TouchController.hpp:

```
1  #ifndef TOUCHCONTROLLER_HPP
2  #define TOUCHCONTROLLER_HPP
3
4  #include <touchgfx/hal/Types.hpp>
5
6  namespace touchgfx
7  {
8   class TouchController
9   {
10   public:
11     virtual ~TouchController()
12     {
13     }
14     virtual void init() = 0;
15     virtual bool sampleTouch(int32_t& x, int32_t& y) = 0;
16  };
17 } // namespace touchgfx
18 #endif /* TOUCHCONTROLLER_HPP */
```

Código 3.32: Clase abstracta TouchController para crear el driver táctil

El driver a implementar debe derivar desde esta clase y dar definición a los métodos virtuales puros *init* y *sampleTouch*. El método *init* ha de iniciar el driver táctil —inicialización de los ADCs encargados de capturar los toques en la pantalla resistiva, inicialización del micro e inicialización de las comunicaciones entre el micro y los ADCs—, mientras que *sampleTouch* ha de devolver, por referencia, las coordenadas de pantalla que han sido presionadas. Se ha de implementar el driver incluso para aquellos casos en los que no se vaya a utilizar el sistema táctil. Por ello, TouchGFX suministra dicho driver, llamado *NoTouchController* que no es más que la derivación de la clase anterior a la que se le ha suministrado cuerpos vacios a las funciones virtuales.

Otra opción es utilizar el driver que internamente usa TouchGFX para la placa STM32F429I-DISCO y su display original de 2,4 pulgadas, y corregir las coordenadas mediante la clase *TouchCalibration*, clase con todos sus métodos estáticos —no se puede crear instancia de clase—.

```
1 #include <touchgfx/transforms/TouchCalibration.hpp>
2 . . .
3 static Point lcdCalibPoints[3]={{48,27},{432,136},{240,245}};
4 static Point touchCalibPoints[3]={{24,32},{216,160},{120,288}};
5
6 TouchCalibration::setCalibrationMatrix(lcdCalibPoints,touchCalibPoints);
```

Código 3.33: Uso de la clase TouchCalibration para corregir las coordenadas táctiles

Para realizar la corrección se deben definir dos tablas de tres puntos —coordenadas «x» e «y» para cada punto—. Una de las tablas ha de contener las coordenadas de pantalla de los puntos a servir de guía para la corrección —línea 3 del código anterior—, mientras que la otra, ha de contener las lecturas que el driver táctil actual devuelve para esos puntos —línea 4—. Habitualmente los puntos de pantalla utilizados son los que se encuentran en las posiciones 10% del ancho y 10% del alto, 90% del ancho y 50% del alto, y 50% del ancho y 90% del alto —datos de la tabla *lcdCalibPoints* correspondiente a la pantalla instalada de dimensión 480x272—. Luego, la corrección es creada mediante la llamada al método estático *setCalibratinMatrix* de la clase *TouchCalibration*. A partir de este momento, las lecturas devueltas por el driver táctil corresponderán con las coordenadas de pantalla para la pulsación realizada sobre la misma.

La librería gráfica, dispone de un driver creado específicamente para el integrado Sitronix ST123x, que utiliza comunicación I2C, pero no es el caso de este proyecto.

Aun habiendo probado el buen funcionamiento de la corrección de coordenadas mediante la clase *TouchCalitration*, se decide realizar el driver desde cero —mediante la derivación de la clase *TouchController*—, ya que supondrá menos potencia de cálculo para el micro. Como se explicó en la sección anterior, el controlador del sistema táctil será el STMPE811 que utilizaba el display original de la placa. Las coordenadas «x» e «y» son devueltas, por este controlador, en cuentas de los ADCs internos, que usa para testar las pulsaciones de pantalla. Por ello se crea un programa específico que implemente la funcionalidad táctil del STMPE811 —a través del archivo de cabecera *stm32f429i_discovery_ioe.h*—, que dibuje nueve cruces en pantalla y muestre sobre la misma las coordenadas, en cuentas de ADC, de las pulsaciones realizadas. Estas cruces están espaciadas, tanto verticalmente como horizontalmente para cubrir las posiciones del 10%, 50% y 90% de la dimensiones de pantalla. El aspecto de lo mostrado en pantalla, por este programa auxiliar, es el siguiente:

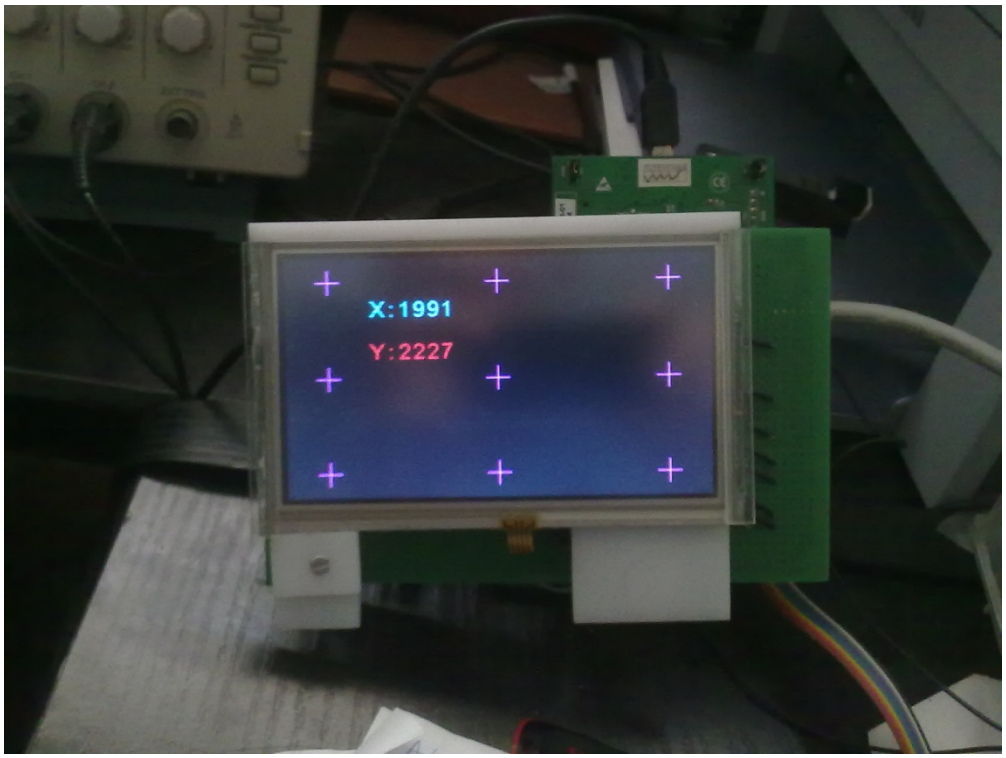


Ilustración 3.41: Aspecto del programa para calibrar el sistema táctil

Con un puntero, se van presionando en cada una de las cruces y anotando las lecturas «x» e «y» de los ADCs del integrado STMPE811 para cada cruz presionada. Con estos datos se crea la siguiente tabla:

Coordenadas Cruz en Pantalla (% de la longitud)		Coordenadas Cruz en Pantalla (pixeles)		Lecturas de los ADCs (cuentas)	
X	Y	X	Y	X	Y
10	10	48	27	3526	807
50	10	240	27	1990	777
90	10	432	27	490	800
10	50	48	136	3535	2198
50	50	240	136	1999	2239
90	50	432	136	495	2200
10	90	48	245	3530	3547
50	90	240	245	2014	3514
90	90	432	245	484	3567

Tabla 17: Lecturas de los ADCs del micro STMPE811 usadas en la calibración táctil

Las primeras dos columnas hacen referencia a las posiciones de las cruces en tantos por ciento de la dimensiones de pantalla —horizontalmente «x» y verticalmente «y»; las dimensiones de pantalla en pixeles son de 480x272—. Cada fila de estas dos columnas representa cada una de las cruces, y cada cruz aparece en la tabla

de arriba hacia abajo correspondiendo al recorrido de izquierda a derecha y de arriba abajo en la pantalla. Las dos siguientes columnas son la realización de estos tantos por ciento expresados en pixeles, y las dos últimas columnas, las lecturas en cuentas de los ADCs internos del circuito extensor de entradas/salidas —STMPE811—.

Para realizar la conversión entre cuentas y pixeles, se procede a generar dos regresiones —una para la coordenada «x» y otra para la coordenada «y»—, donde se relaciona cada valor de pantalla en pixel, para cada cruz, con las cuentas correspondientes del ADC. Cada regresión consta de nueve puntos, idealmente solo tres de ellos son diferentes. Las gráficas de estas dos regresiones son las siguientes:

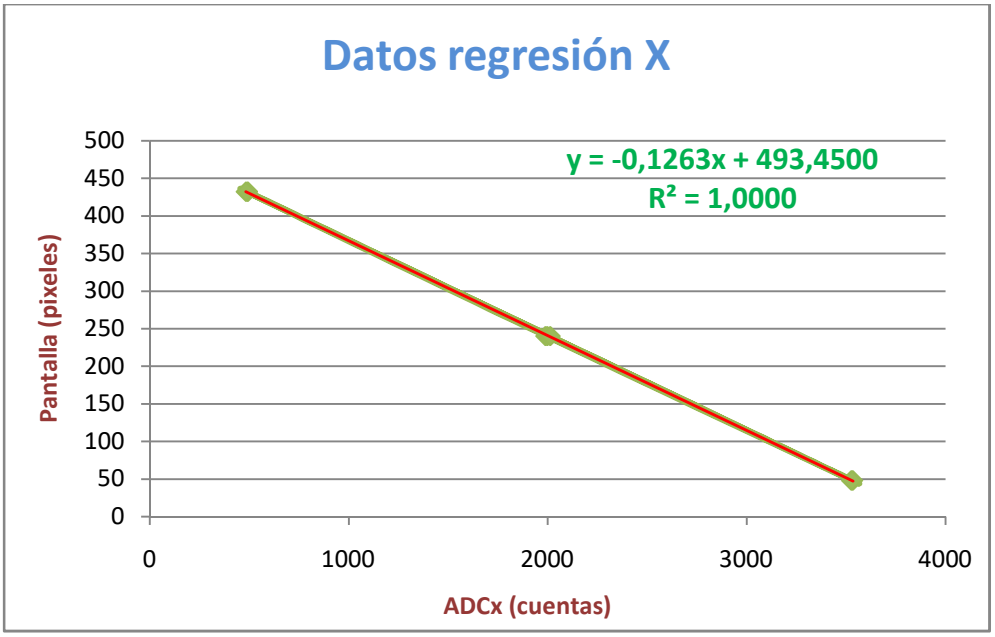


Ilustración 3.42: Gráfica de la regresión de la coordenada «x» para la calibración táctil

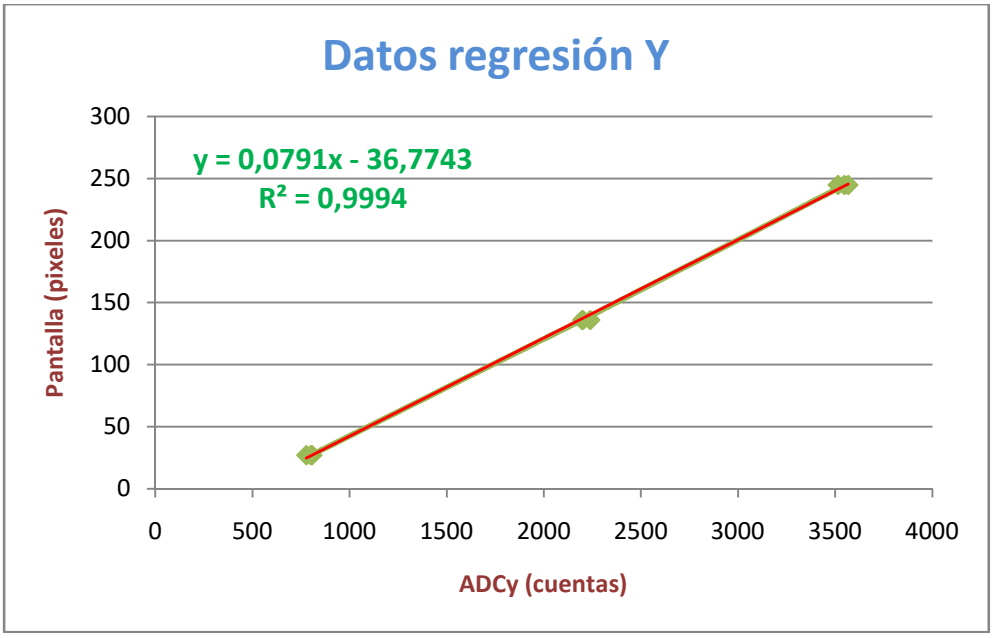


Ilustración 3.43: Gráfica de la regresión de la coordenada «y» para la calibración táctil

En ambos casos el coeficiente de determinación —erre cuadrado— es muy próximo a uno —en el caso de la regresión en «x» es uno—, con lo que se puede afirmar que el modelo lineal es suficiente para relacionar cuentas con pixeles. En cada gráfica, además del coeficiente de determinación, aparece la función de la recta de regresión, donde la variable independiente «x» representa el dato de cuentas de los ADCs, independientemente si se trata de la coordenada horizontal —primera gráfica— o vertical —segunda gráfica—, y la variable dependiente «y» que representa la traducción de esas cuentas en pixeles. Con estas dos funciones de regresión —para cada coordenada—, y a partir de la derivación de la clase que se mostraba en el Código 3.32, se desarrolla la clase que representa el driver para el sistema táctil de la nueva pantalla instalada. La definición de esta clase puede ir en un archivo de cabecera que sea incluido en el archivo BoardConfig.cpp, pero se decide incluirla dentro de este, en la zona del nombre de espacio touchgfx, para que la capa HAL —dentro de este nombre de espacio— no tenga problemas en usarla. La definición de este driver queda finalmente así:

```
. . .
class ResistiveTouchControllerJC : public TouchController
{
    private:
        uint32_t _x, _y;

    public:
        virtual void init()
        {
            _x=0;
            _y=0;
            IOE_Config();
        }

        virtual bool sampleTouch(int32_t& x, int32_t& y)
        {
            uint32_t xDiff, yDiff , x_i , y_i;
            bool tocado=false;
            tocado = (I2C_ReadDeviceRegister(IOE_REG_TP_CTRL) & 0x80);
            if (tocado)
            {
                x_i = I2C_ReadDataBuffer(IOE_REG_TP_DATA_X);
                y_i = I2C_ReadDataBuffer(IOE_REG_TP_DATA_Y);
                x_i=-0.1263*x_i+493.4500 ;
                y_i=0.0791*y_i-36.7743;
                xDiff = x_i > _x? (x_i - _x): (_x - x_i);
                yDiff = y_i > _y? (y_i - _y): (_y - y_i);
                if (xDiff + yDiff > 10)
                {
                    _x = x_i;
                    _y = y_i;
                }
            }
            I2C_WriteDeviceRegister(IOE_REG_FIFO_STA, 0x01);
            I2C_WriteDeviceRegister(IOE_REG_FIFO_STA, 0x00);
            x=_x;
            y=_y;
            return tocado;
        }
};
. . .
```

Código 3.34: Clase que representa el driver para el sistema táctil, parte de BoardConfig.cpp

La clase es *ResistiveTouchControllerJC* que deriva de la clase abstracta *TouchController* —antes mostrada—. Como se mencionaba, se han de dar cuerpo a las funciones miembro *init* y *sampleTouch*, pero pueden crearse nuevos miembros si ello es útil. De hecho, para esta clase, se crean las variables miembro privadas «*_x*» y «*_y*», que servirán para almacenar los valores de las coordenadas de la pulsación anterior a la actual, con el objeto de poder implementar un mecanismo que evite, en lo posible, la variabilidad de las coordenadas pulsadas cuando se mantiene la pulsación o cuando se procede a realizar un arrastre sobre pantalla. El cuerpo de la función *init*, consiste en la inicialización a cero de estas dos variables miembro y la llamada a la función *IOE_Config* —declarada en el archivo de cabecera *stm32f429i_discovery_ioe.h*, que está incluido en el archivo *BoardConfig.cpp*, donde está definida esta clase—. Esta función —*IOE_Config*—, como ya fue comentado en la sección anterior 3.5.1.6, inicializa el integrado STMPE811 para poder gestionar una pantalla táctil resistiva, así como inicializar las funciones del micro que permitan comunicarse con este periférico externo —instalado en la placa STM32F429I-

DISCO—. Con ello, la función *init* ya tiene la funcionalidad requerida y la capa HAL puede llamarla con la seguridad que se van a realizar todas las inicializaciones necesarias en cuanto a driver táctil se refiere. Para el cuerpo de la función *sampleTouch*, se crean las variables internas «x_i» e «y_i», que serán las encargadas de contener las coordenadas actualmente pulsadas, las variables «xDiff» e «yDiff», que contendrán los valores absolutos de las diferencias entre las coordenadas actualmente pulsadas respecto a las anteriores, y la variable «tocado», que indicará si realmente se ha producido una pulsación sobre la pantalla —ya que *TouchGFX* va a comprobar las pulsaciones realizadas mediante consulta permanente, llamando a esta función en cada tick del sistema gráfico—. La determinación de si se ha producido pulsación o no, se realiza mediante la llamada a la función *I2C_ReadDeviceRegister* —línea 19—, perteneciente a la librería *stm32f429i_discovery_ioe.h,* ya comentada en la sección anterior 3.5.1.6. Si se ha producido la pulsación se procede a realizar las lecturas de las coordenadas pulsadas —líneas 22 y 23— mediante la llamada a la función *I2C_ReadDataBuffer* —comentada igualmente en la sección anterior— con los argumentos pertinentes para obtener cada coordenada. Luego, se procede a aplicar las funciones de regresión —líneas 24 y 25— que convierten las lecturas, obtenidas con la función anterior, desde cuentas a coordenadas en pixeles de pantalla. Si la diferencia entre las presentes coordenadas, obtenidas y tratadas, respecto a las anteriores, es mayor a 10 pixeles, se dan por válidas y se asignan como coordenadas anteriores para ser tenidas en cuenta en pulsaciones futuras —líneas 30 y 31—. Estos son los valores de coordenadas devueltas por referencia en el argumento de la llamada a esta función —líneas 36 y 37—. En el caso de que los nuevos valores de coordenadas, obtenidos y tratados, no superen la diferencia establecida con los anteriores, no se produce a la actualización de los valores anteriores y en consecuencia, son los valores de la pulsación anterior los retornados por referencia. En cualquier caso, la llamada a la función retorna el valor lógico que representa si se ha realizado o no la pulsación, para informar de ello a TouchGFX, como se mencionó anteriormente.

Una vez creada la clase que representa el driver táctil, es necesario crear un objeto de esta clase y registrarlo en el objeto que representa la capa HAL, para que esta pueda gestionarlo. Esto se produce, al igual que la definición del driver táctil, en el archivo de configuración BoardConfig.cpp. El detalle de cómo se realiza será comentado en la sección 3.5.2.

3.5.2 Inicialización software: Función touchgfx_init().

Antes de proceder a la declaración de la función *touchgfx_init*, que inicializa el software de *TouchGFX*, se han de crear una serie de objetos que se van a necesitar, estos objetos son: la instancia del driver DMA, la instancia del driver de la pantalla táctil, la instancia del display y la instancia del objeto instrumentación. Estos cuatro objetos son creados en el archivo *BoardConfiguration.cpp* dentro del espacio de nombres *touchgfx*. Sus declaraciones son las siguientes:

```
1  namespace touchgfx
2  {
3      . . .
4      STM32F4DMA dma;//Instancia del driver DMA
5      ResistiveTouchControllerJC tcJC;//Instancia del driver táctil
6      LCD16bpp display;//Instancia del driver de pantalla
7
8      CortexMMCUInstrumentation mcuInstr;//instancia de la clase instrumentación del micro
9      . . .
10 }
```

Código 3.35: Declaración de objetos para la capa HAL en el archivo BoardConfiguration.cpp

En la línea 4 aparece la declaración del objeto *dma* de la clase STM32F4DMA. Esta clase es una especialización de la clase abstracta DMA —que no admite instancia de clase—, y representa el driver para el DMA en *TouchGFX*. En esta clase derivada se sobrescriben varios métodos, uno de los más importantes es *setupDataCopy,* cuyo cometido es configurar el hardware DMA —en el caso concreto se trata del periférico DMA2D (transferencia DMA bidimensional)— para cada transferencia e iniciarla conforme a los datos de una estructura —de tipo *BlitOp*— pasada como referencia en su argumento. Los datos contenidos en esta estructura van desde el tipo de operación a realizar—como simple copia de datos o copia con información alfa—, hasta información referente al área rectangular a transvasar. Existen otros métodos que han de sobrescribirse en la clase derivada, pero, ya que la clase especializada es la propia del micro utilizado, no se ha realizado ningún desarrollo sobre este tema y por tanto, no es el objeto de este proyecto.

En la línea 5 se declara el driver de la pantalla táctil. En el caso de este driver sí se ha requerido desarrollo, ya que se ha instalado un nuevo display en la placa utilizada —STM32F49I-DISCO—, soportada directamente por TouchGFX, pero la pantalla táctil que viene incorporada al nuevo display requiere de un driver para que la pueda gestionar el entorno gráfico —ya que se debe establecer una nueva correspondencia entre coordenadas pulsadas y zona del display afectada—. En esta línea se crea el nuevo driver de la clase *ResistiveControllerJC*, cuyo desarrollo ha sido explicado en la sección 3.5.1.7.

El siguiente driver instanciado es el referente al display —línea 6—. Este es un driver genérico de tipo RGB565 —16 bits—, y en la versión de TouchGFX tratada —4.2—, es el único disponible —la última versión, la 4.8, dispone, además, de un driver de display de 24 bits—. Es genérico en el sentido que no está especificado para un tamaño concreto de pantalla, el tamaño se especifica al crear el objeto HAL al agregar los drivers al mismo. Hay que aclarar que este driver no gobierna la actuación del display directamente, actúa sobre los búferes de display; el control sobre el display es realizado mediante el periférico LTDC —para este proyecto, ya que se ha usado un micro del fabricante ST— y ya ha sido configurado en el cuerpo de la función *LCD_Config* que es llamada en la función de inicialización hardware *hw_init*.

El último driver creado es el de instrumentación del micro. Se trata de la instancia *mcuInstr* de la clase *CortexMMCUInstrumentation*, que es derivada de la clase *MCUInstrumentation*, y como en los otros drivers, se ha de dar cuerpo a varios métodos virtuales de la misma. Su cometido es medir el lapso de tiempo, expresado en ciclos de reloj, entre dos momentos dados. Su método más importante es *setMCUActive*, que debe ser llamado por la tarea por defecto del sistema operativo —tarea idle— y esta ha de pasarle como argumento el dato booleana que indique si está activo el micro o si no lo está. Todo ello fue descrito en la sección 2.5.5.

En la sección 1.2.1 se mostró un esquema de capas donde la capa HAL se presentaba como un ente dependiente del micro utilizado y configurable mediante la inserción de varios módulos que dependían de la placa.

Estos módulos son los driver que se acaban de instanciar y esquemáticamente se pueden representar como se muestra en la siguiente ilustración:

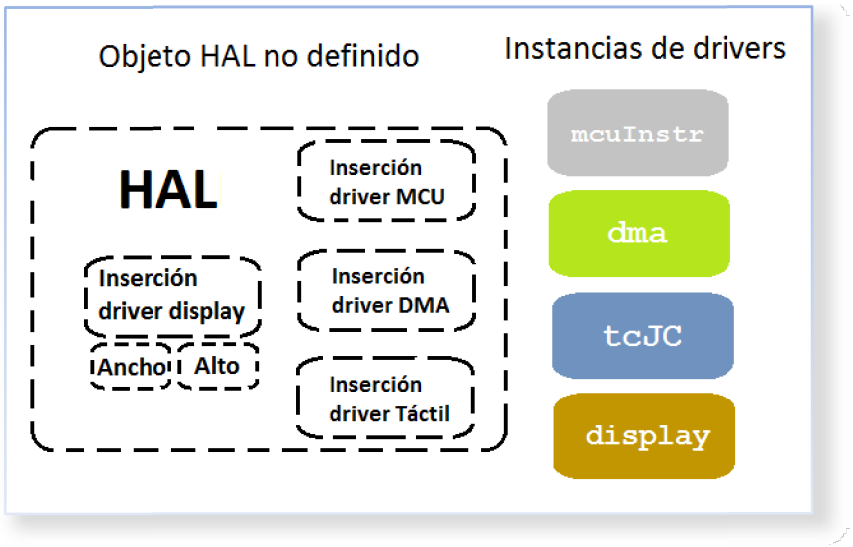


Ilustración 3.44: Esquema de las instancias de los drivers de la capa HAL antes de ser instalados

Con los driver ya declarados, es posible definir el cuerpo de la función de inicialización software de TouchGFX:

```
1  void touchgfx_init()
2  {
3   /* Creación de la capa HAL con instalación de los drivers DMA, pantalla y táctil */
4   HAL& hal = touchgfx_generic_init<STM32F4HAL>(dma, display, tcJC, 480, 272,\
5                                                (uint16_t*)0xd00bf400, 0x000bf400);
6   hal.setFrameBufferStartAddress((uint16_t*)frameBuf0);
7   hal.setTouchSampleRate(1);
8
9   /* Esta sentencia permite manipular simultáneamente accesos de DMA y accesos del
10    TFT en la SDRAM. */
11  hal.lockDMAToFrontPorch(false);
12
13  /* Se inicia el objeto instrumentación del micro */
14  mcuInstr.init();
15
16  /* El objeto instrumentación es añadido a la capa HAL y se habilita su funcionalidad */
17  hal.setMCUInstrumentation(&mcuInstr);
18  hal.enableMCULoadCalculation(true);
19 }
```

Código 3.36: Definición de la función de inicialización de software — touchgfx_init—

Esta función, al igual que la de inicialización hardware —sección 3.5.1—, es llamada desde la función main para inicializar el sistema. Mediante la llamada a *touchgfx_generic_init* —función plantilla entre las líneas 4 y 5—, se crea el objeto HAL —se crea de forma estática internamente, ya que solo devuelve su referencia—, y se insertan los drivers de DMA, pantalla táctil y display, especificando el tamaño de este último. Los dos últimos parámetros hacen referencia a la dirección de memoria donde albergar bitmaps y su tamaño, respectivamente —ver sección 3.5.1.2—. A continuación se especifica la dirección de memoria donde se encuentran los buffer gráficos —línea 6—. La dirección está contenida en la variable *frameBuf0* y está declarada al inicio del archivo *BoardConfiguration.cpp*, su valor es el de la dirección de inicio de la memoria RAM externa. En la llamada a la

función que aparece en la línea 6 del código anterior, se están omitiendo dos argumentos por defecto, que indican a la función si usar un buffer secundario y un tercer buffer —para animaciones—. Estos argumentos son de tipo booleano y sus valores por defecto son verdaderos, por lo que sí se van a utilizar estos buffers adicionales. El tamaño de los tres buffers serán iguales y *TouchGFX* los determina mediante la información pasada en la función de la línea 4, es decir de la altura y anchura del display y del tamaño en bytes de cada punto de pantalla —es de dos bytes ya que el driver del display es de formato RGB565—. El primer buffer estará localizado en la dirección indicada por *frameBuf0*, el segundo en la dirección que resulte de sumarle a la anterior el tamaño del buffer y el tercero, de sumarle a la dirección inicial dos veces el tamaño del buffer. Seguidamente —línea 7—, se instruye al entorno gráfico que ha de testear los eventos táctiles en cada actualización del display, y que el objeto DMA podrá acceder a la memoria RAM a la vez que lo hace el periférico interno LTDC —línea 11— . Finalmente, se inicializa el driver de instrumentación —línea 14—, que consistirá en habilitar las capacidades de depuración del micro y la cuenta de ciclos, se agrega al objeto HAL —línea 17— y se activa las capacidades de mediad de este driver —línea 18—. Esquemáticamente la capa HAL se puede representar así:

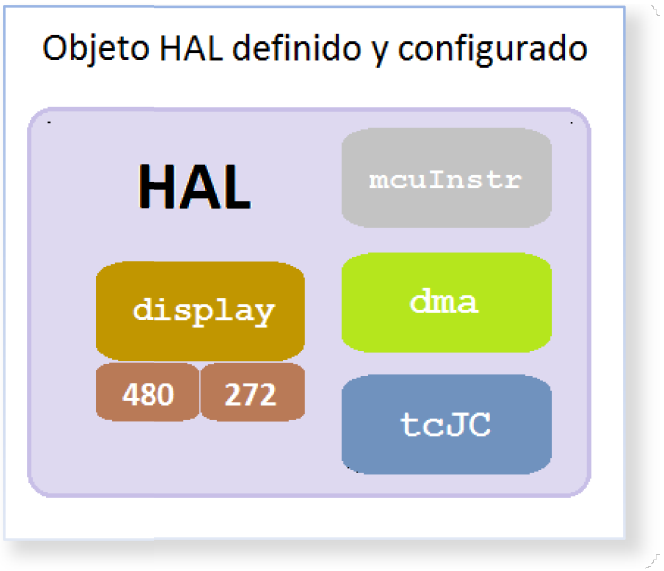


Ilustración 3.45: instancia de la capa HAL configurada

3.5.2.1 Secuencia de inicialización.

La secuencia de inicialización ocurre en la función *main* dentro del archivo main.cpp:

```
1  int main (void)
2  {
3      hw_init();
4      touchgfx_init();
5      configuraGenerador();
6      configuraPrueba();
7      fft.configuraHardware(1);
8
9      . . .
10 }
```

Código 3.37: Inicialización del sistema

Primeramente se ha de configurar el hardware que *TouchGFX* utilizará directamente. Esto se realiza mediante la llamada a la función *hw_init* —descrita en la sección 3.5.1—. Luego, se ha de configurar el software consistente en el propio TouchGFX, mediante la llamada a la función *touchgfx_init* —descrita en esta misma sección 3.5.2—. Luego se realizan configuraciones de hardware que no tienen nada que ver con TouchGFX, como es la configuración de las señales de prueba, por parte del DAC del micro —mediante la llamada a la función *configuraGenerador* que es descrita en la sección 4.3—, la configuración de una señal cuadrada, que verifica la correcta configuración del reloj —mediante la llamada a la función *configuraPrueba*— y la configuración del hardware propio de la aplicación útil —la que se encarga de adquirir los datos y calcular la transformada rápida e Fourier—, mediante la llamada al método *configuraHardware* del objeto *fft* que representa el objeto de la aplicación —todo ello descrito en la sección 4.5—.

4 Elaboración del software del proyecto

4.1 Introducción

Antes de iniciar la creación del entorno gráfico, mediante la programación de líneas de código, o la de nuevos componentes —*Widgets*—, es necesario establecer la estructura gráfica que tendrá este entorno, definiendo las partes en las que se divide, así como qué componentes gráficos serán necesarios y dónde estarán localizados. Es cierto que las imágenes que se muestran a continuación han sido hechas después de la realización de todo el programa, pero una maquetación inicial, más tosca y con esquemas hechos a mano, fue realizada de forma similar.

La apariencia general de la única vista utilizada, es la siguiente:

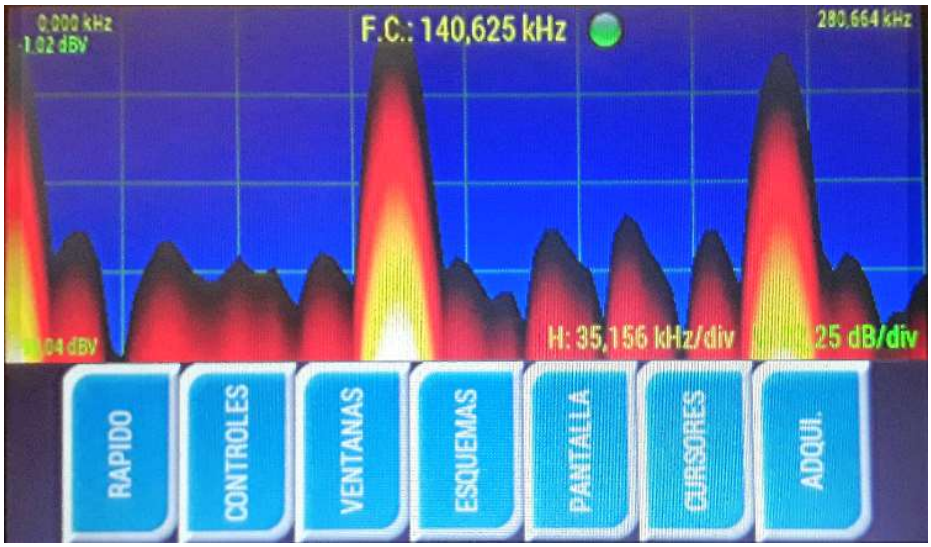


Ilustración 4.1: Apariencia gráfica de la vista

Esta está dividida en dos partes: la zona de la gráfica, y la zona de los menús. Tanto la zona de la gráfica, como la de los menús, ocupan horizontalmente toda la pantalla. En cambio, la gráfica ocupa, verticalmente, los dos tercios superiores, mientras que los menús, el tercio inferior. El fondo de la gráfica se rellena con color azul oscuro y en ella se localizan una serie de controles que están relacionados con la representación gráfica del espectro, informando o controlando algún parámetro. Alguno de estos controles son de nueva creación,

específica para este proyecto. El fondo de la zona de los menús se rellena con color morado oscuro, y sobre ella se localiza el menú constituido por los elementos de menú que permiten controlar la funcionalidad de toda la aplicación. Los elementos de menú consisten en unos botones de enclavamiento que se quedan en estado presionado cuando son tocados, para volver a su estado de reposo cuando se les vuelve a tocar. En su estado de enclavamiento se desplazan hacia la izquierda de la zona de los menús, desplegando un contenedor que alberga los controles necesarios correspondientes a la funcionalidad concreta del elemento de menú seleccionado. Estos elementos de menú, tal y como se muestra en la figura precedente, son «Rápido», «Controles», «Ventanas», «Esquemas», «Pantalla», «Cursores», y «Adquisición». A partir de este momento, y para el resto del texto de este escrito, se utilizará de forma indistinta «elemento de menú» o directamente «menú», ya que esta última es más corta y realmente cada uno de estos elementos constituye un menú donde se seleccionan opciones de funcionamiento o configuración. Tanto la clase de cada elemento de menú, como la clase que engloba a todos ellos, serán componentes gráficos de nueva creación —ya que la versión inicial de *TouchGFX*, con la que se empieza a trabajar, no dispone de controles similares—. La estructura de cada menú será descrita más adelante.

En la zona de la gráfica se localizan los siguientes elementos:

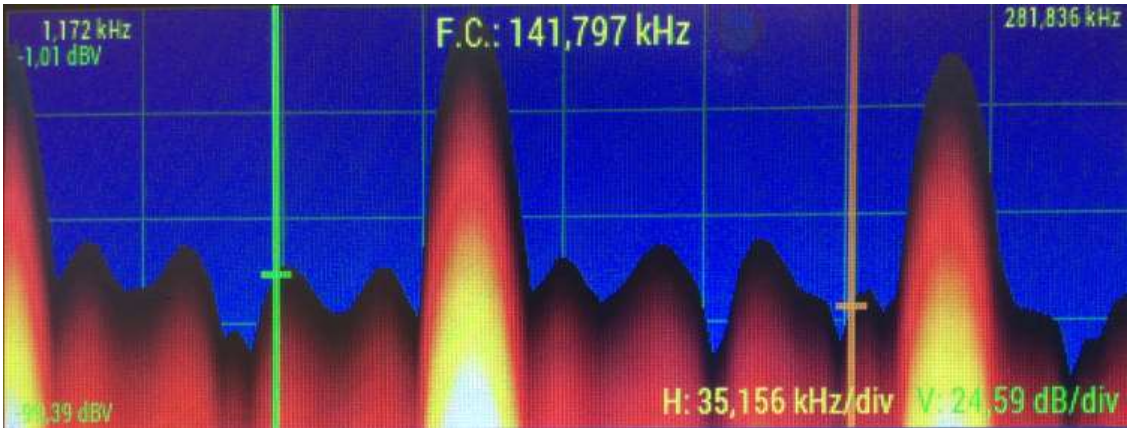


Ilustración 4.2: Apariencia de la zona de la gráfica de la vista

El control más importante es la gráfica —el que representa el espectro—, pero también hay otros controles como son: los textos informativos, la rejilla —retícula—, los cursores, el objeto marcador del estado de la adquisición, el área invisible que ocupa toda la zona de la gráfica, encargada de recoger los eventos necesarios para producir el desplazamiento de la misma cuando se produce el arrastre sobre ella, así como mostrar los botones de escalado de los ejes cuando se produce un toque sobre esta área. A excepción de los textos informativos, todos los controles descritos son de nueva creación, y su descripción será realizada a lo largos de este capítulo. Los textos informativos se dividen en aquellos referentes a frecuencia y a aquellos que lo son al nivel. Así los textos que aparecen en la parte superior de la ilustración, en color amarillo, informan, de izquierda a derecha, de la menor frecuencia representada en pantalla, de la frecuencia en el centro de la misma, y la mayor frecuencia representada. En la zona inferior de la pantalla, hacia la derecha y en color amarillo, igualmente, se localiza el texto que informa cuanta frecuencia representa una división horizontal. Los textos informativos de nivel tienen color verde y los que representan el mayor y menor nivel en pantalla, se localizan en la zona izquierda de la misma. En la zona inferior derecha, está el texto informativo del nivel por división vertical. Los textos son los controles que están encima del resto. Los cursores están encima de la gráfica y la rejilla puede estar tanto atrás del gráfico como delante —ya que realmente existen dos rejillas; una delante y otra atrás,

ocultándose la que no se utiliza—. La apariencia de la zona de la gráfica respecto a la representación de la rejilla, es la siguiente:

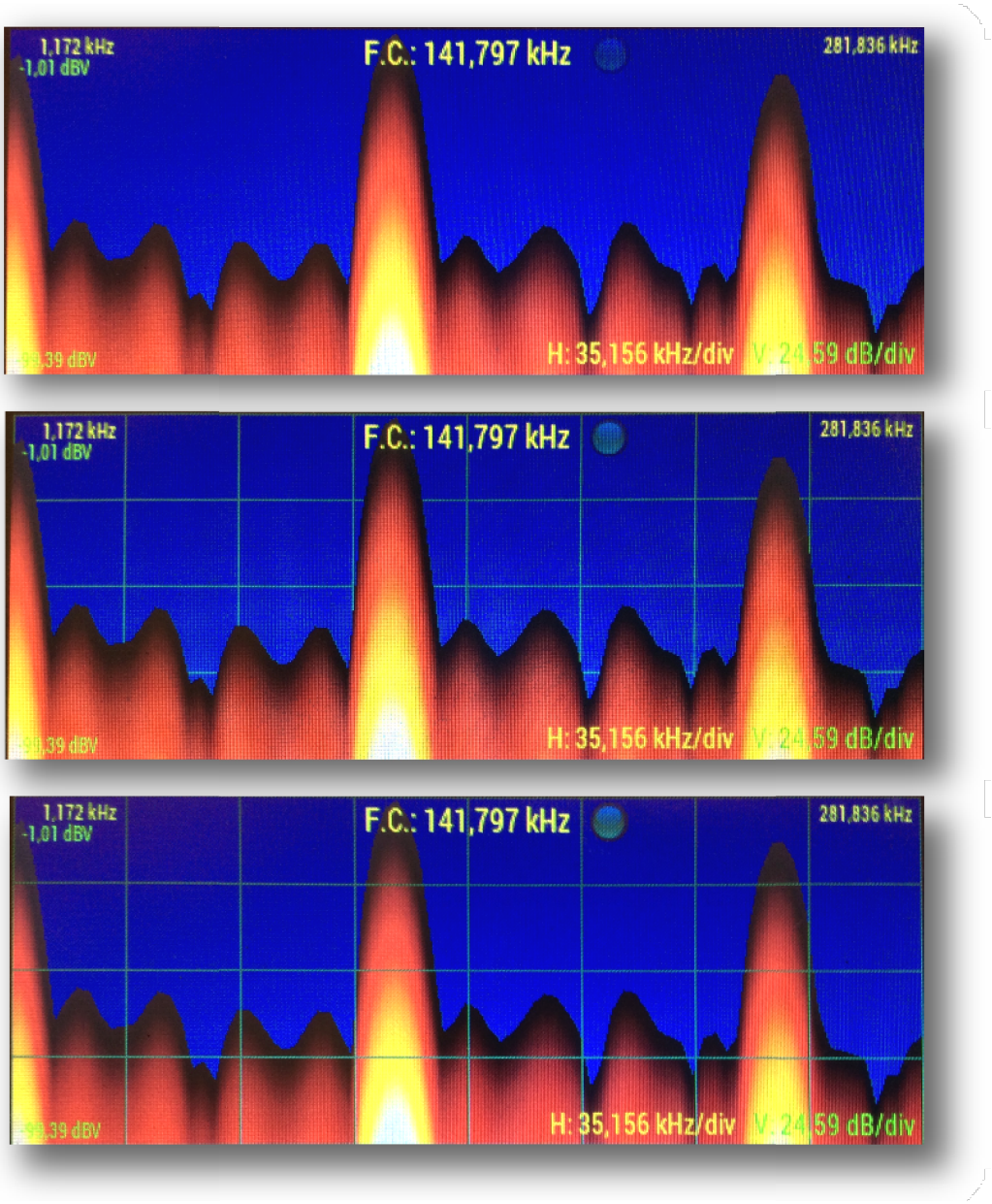


Ilustración 4.3: Diferente posiciones de la retícula en la zona de la gráfica de la vista

Tal y como se muestra en esta figura, la gráfica puede estar sin rejilla —imagen superior—, la rejilla estar detrás de la gráfica —imagen central— o estar delante —imagen inferior—.

A continuación se procede a describir los menús. El menú «Rápido» tiene por cometido servir de método rápido de modificación de parámetros contenidos en otros menús. Su estructura es la mostrada en la siguiente figura:

Ilustración 4.4: Apariencia del menú Rápido

Al desplegar su contendor, este menú presenta unos controles denominados «botones de estado» —controles de nueva creación, ver sección 4.4.1.8—. Estos botones constan de varias opciones que son cambiadas, de forma cíclica, al ser presionados de forma continuada, modificando una propiedad concreta de la aplicación. Así el botón de estado «VENTANAS» cambia la ventana utilizada en la FFT entre la posibles ventanas existentes —ver secciones 4.5.3 y 4.5.6—; el botón «LOGARITMICO», permite cambiar entre representación logarítmica o lineal; «AUTIO», realiza un ajuste en la gráfica de su nivel para que ocupe toda la pantalla; «SEGUIM.» —seguimiento—, hace exactamente lo mismo que el botón anterior pero manda realizar un nuevo ajuste en cada actualización de pantalla —es un autoajuste continuado—; «CANAL», elige el origen de los datos capturados y posteriormente procesados, entre el canal 1 o el 2; «ESQUEMAS», selecciona la gama de color relleno entre la gráfica y el eje horizontal —ver sección 4.4.1.11—; y «RES. FRECUEN.», permite seleccionar la resolución de frecuencia de muestreo entre la cuatro posibles —ver sección 4.5.5—. Todos estos botones de estado no pueden ser mostrados a la vez debido a sus dimensiones —esta es la razón por la que se muestran dos imágenes en la ilustración anterior—, por ello, están alojados en un contenedor que solo permite mostrar, longitudinalmente, una distancia equivalente a cuatro de estos botones. Para poder acceder a los no mostrados, se ha de presionar y arrastras, sobre este contenedor, permitiendo mostrar el botón de estado deseado. Para indicar que se debe realizar el arrastre a izquierda o derecha de dicho contenedor, para poder mostrar todos los botones, existen en sus laterales dos grandes flechas parpadeantes. Estas son controles de tipo «marcador», un nuevo componente desarrollado para este proyecto —ver sección 4.4.1.3—.

El menú «CONTROLES» dispone de cuatro contenedores. Se pasa de uno a otro mediante el arrastre con el dedo u otro objeto que realice presión. En la parte inferior de este menú aparecen cuatro puntos que indican qué contenedor se está visualizando en cada momento. El primer contenedor contiene los controles que informan y modifican la frecuencia que se encuentra en la mitad de la pantalla; en el segundo, los referentes al ancho de

frecuencia mostrado en pantalla; en el tercero, los relativos al mínimo nivel mostrado en pantalla; y en el cuarto, cuanto nivel se representa en el eje vertical. Su apariencia se muestra en la siguiente ilustración:

Ilustración 4.5: Apariencia del menú Controles

Todos los contenedores del menú «CONTROLES» tienen dos textos: uno que indica sobre que parámetro se informa —en la parte superior del contenedor, en letra pequeña y centrado horizontalmente—, y otro que indica el valor del parámetro actual —centrado tanto horizontalmente como verticalmente, mostrado en letra grande— . Para modificar estos parámetros, aparece en cada contenedor dos botones a los lados del valor del parámetro. En todos los contenedores estos botones tienen la misma apariencia, pero no se tratan del mismo control. En los contenedores para los parámetros de «Frecuencia Central» y «Posición Vertical», se trata de controles de tipo botones de retención —*HoldableButton*— una clase no suministrada directamente en la librería *TouchGFX*, y es obtenida desde el foro de discusión de la web del fabricante de *TouchGFX*. Este tipo de control tiene la particularidad de generar eventos de presión de forma repetida mientras se mantienen presionados. En cambio, los botones que aparecen en contenedores «Frecuencia Span» y «Escala Vertical», son botones ordinarios —*Buttons*—.

El siguiente menú es «VENTANAS», referente a la selección de las ventanas temporales utilizadas en la FFT. Su apariencia es la siguiente:

Ilustración 4.6: Apariencia del menú Ventanas

Este menú solo dispone de un contenedor, donde se localizan cinco botones de selección mutuamente excluyentes —*RadioButton*—, que permiten seleccionar una de las cinco ventanas temporales disponibles —ver sección 4.5.3—. Este menú también dispone de un botón que permite el autoajuste.

El siguiente menú es «ESQUEMAS», que posibilita la selección de las distintas gamas de color entre la gráfica y el eje horizontal:

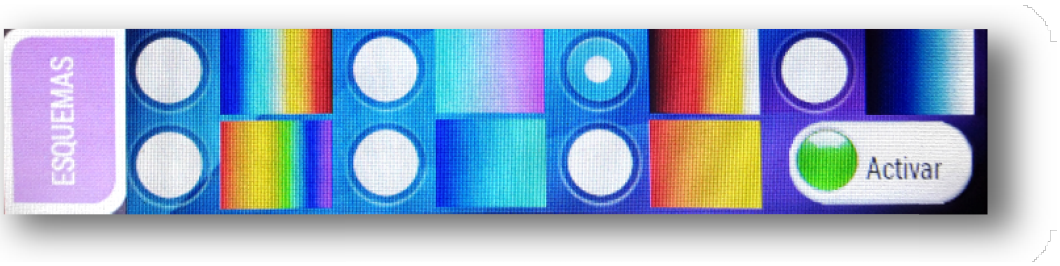


Ilustración 4.7: Apariencia del menú Esquemas

Al igual que el menú anterior, dispone de botones de selección —siete botones— que posibilitan la selección de los distintos esquemas y la representación, o no, de estas, mediante un botón de selección —Activar—.

El menú «Pantalla», cuenta con tres contenedores, uno dedicado a la traza de la gráfica, otro dedicado a la posición de la rejilla y color de fondo y otro dedicado a los textos en pantalla. En el primer contenedor hay dos barras de deslizamiento —controles extraídos de los códigos ejemplo de *TouchGFX*—, una dedicada al grosor de la traza de la gráfica, y otra a su color. Este contenedor también dispone de un botón de selección que permite la representación de la gráfica en forma de puntos —datos tratados— o vectores —unión de los puntos mediante segmentos—. El segundo contenedor dispone de dos barras de deslizamiento que permiten variar el color y la luminosidad del fondo del gráfico, y de dos botones de selección que permiten controlar la posición de la rejilla y si visibilidad. El tercer contenedor tiene dos barras de deslizamiento, cada una para controlar el color de los

textos de frecuencia y los textos de nivel, así como sendos botones que controlan la visibilidad de estos. La apariencia de este menú es la siguiente:

Ilustración 4.8: Apariencia del menú Pantalla

El siguiente menú, «Cursores», tiene igualmente, tres contenedores. En el primero de ellos se tienen dos ruedas de selección —componentes de tipo JogWheel, extraídos de códigos ejemplo de TouchGFX—, que controlan la posición horizontal de dos cursores, y once textos que informan de las lecturas de estos cursores. En el segundo contenedor aparecen tres barras de desplazamiento, una dedicada al color del curso1, otra dedicada a su transparencia, y otra a su luminosidad. El tercer contenedor es exactamente igual al segundo pero dedicado a controlar la apariencia del cursor2.

La apariencia de este menú, es la siguiente:

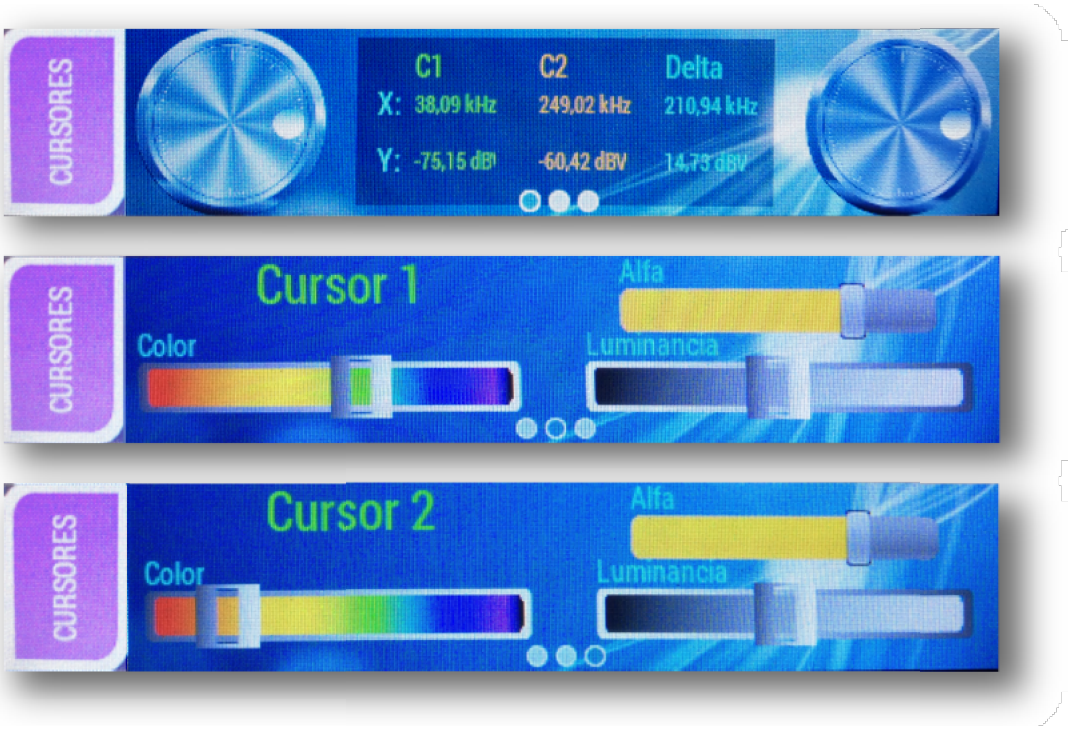


Ilustración 4.9: Apariencia del menú Cursores

Finalmente está el menú «Adquisición», con un control selector con elementos de selección que permiten modificar la frecuencia de muestreo —entre cuatro posibles, ver sección 4.5.5—; dos botones de opción que permiten, respectivamente, detener y reanudar la adquisición y mostrar los datos en el dominio de la frecuencia o del tiempo; dos botones de opción que permiten la selección del canal de origen —entre canal 1 y canal2—; y otros dos botones de opción para, respectivamente, elegir entre representación logarítmica y línea, y si se realiza el autopan al realizar el autoajuste —autoajuste hecho exclusivamente a los datos mostrados en pantalla y no a todo el registro—. La apariencia de este menú es la siguiente:

Ilustración 4.10: Apariencia del menú Adquisición

4.2 Configuración del reloj del sistema

Una de las configuraciones más importantes a tener en cuenta es la del reloj del sistema, que determina la frecuencia de funcionamiento del núcleo del micro así como los relojes derivados para el resto de periféricos. La librería compilada de *TouchGFX* ya trae consigo el archivo de configuración del reloj del micro. Este archivo es system_stm32F4xx.c, perteneciente a la librería estándar de periféricos del fabricante ST, pero es inaccesible, ya que como se ha comentado, está compilado dentro de la librería gráfica. La frecuencia, por defecto, a la que trabaja el núcleo del micro, su máxima posible, es de 180 MHz. *TouchGFX* configura el reloj del micro para trabajar a esta frecuencia. El esquema para la configuración del reloj, extraída del manual del fabricante, es la siguiente [4]:

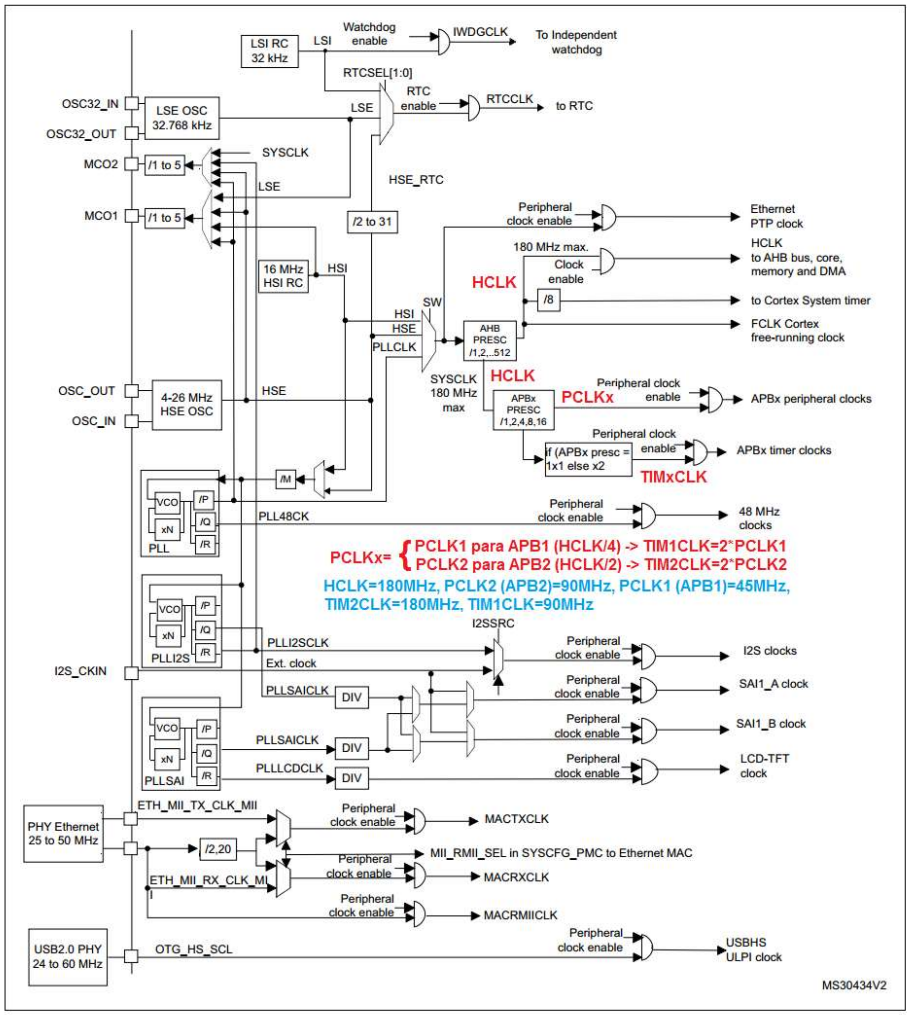


Ilustración 4.11: Esquema del reloj del micro STM32F429

El sistema del reloj dispone de varias opciones como fuente de señal, la más habitual, y la que utiliza la placa DISCO, es la procedente de los terminales OSC_IN y OSC_OUT —pines de puerto PH0 y PH1, pines del micro 23 y 24, oscilador HSE—, donde se conecta el cristal de cuarzo de 8 MHz. Esta señal de reloj pasa por el PLL principal donde es convertida a una frecuencia mayor —180 MHz— constituyendo la señal PLLCLK mostrada en el

esquema, situada a la entrada del multiplexor que selecciona la fuente de reloj. A la salida de este, la señal pasa a ser llamada SYSCLK —180 MHz—. Esta señal se bifurca en varias, para disciplinar varios subsistemas internos del micro con distintas frecuencias de trabajo. Los relojes más importantes, y tenidos en cuenta en este proyecto en el cambio de la configuración por defecto, son los referentes al que disciplina el núcleo —HCLK a 180MHz por defecto— y los que lo hacen a los dos buses avanzados de periféricos —APB—, esto es, las señales de reloj PCLK1 —45MHz por defecto para APB1— y PCLK2 —90MHz por defecto para APB2—. Como será tratado más adelante, a la hora de configurar los ADCs, la frecuencia máxima a la que pueden trabajar es de 36 MHz, que asegurará su máxima velocidad de muestreo —ver sección 4.5.5—. Los periféricos ADCs están conectados al bus APB2, bus que funciona a la mitad de frecuencia del reloj HCLK, y el reloj de los ADCs es obtenido a través del reloj PCLK2 —bus APB2— dividido por un factor, el menor de los cuales es de 2. Por tanto, si se persigue una frecuencia de trabajo de los ADCs de 36 MHz, se necesita un reloj máximo PCLK2 de 72 MHz, y en consecuencia un reloj HCLK de 144 MHz. Es decir, que se ha de bajar la rapidez del núcleo del micro —y del resto de subsistemas de este—, para poder obtener la máxima velocidad de muestreo posible.

Sin embargo existe un problema, no se tiene acceso al archivo *system_stm32F4xx.c* —compilado dentro de *TouchGFX*—. La solución que se encuentra es incluir en el proyecto del compilador —*Keil*— este archivo fuente, extraído de la librería estándar de periféricos de ST —no compilada—, tal y como se muestra en la siguiente ilustración:

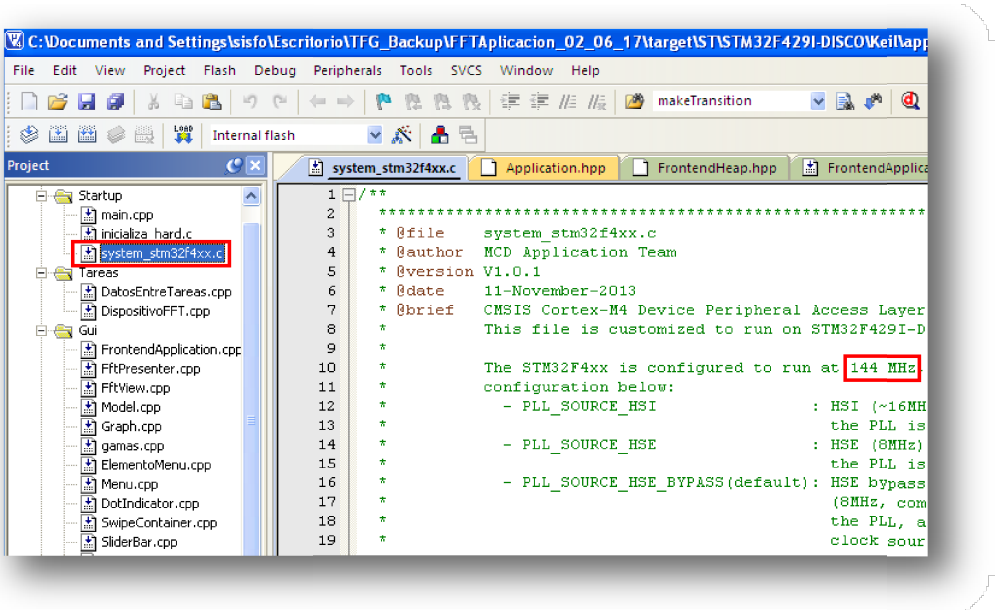


Ilustración 4.12: Archivo system_stm32f4xx.c para modificar el reloj del sistema

Ahora es necesario modificar las constantes *PLL_M*, *PLL_N* y *PLL_P* del archivo *system_stm32xx.c* —valores M, N y P del PLL principal, mostrado en la primera ilustración en esta sección— para obtener un PLLCLK de 144 MHz y un PCLK2 de 72 MHz. La parte de este archivo que ha sido modificada es la siguiente:

```
1  #define PLL_SOURCE_HSE // HSE (8MHz) used to clock the PLL, and the PLL is used as system
2
3  /* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
4  #if defined  (PLL_SOURCE_HSI)
5  #define PLL_M      16
6  #else
7  #define PLL_M      8
8  #endif
9  #define PLL_N      288
10
11 /* SYSCLK = PLL_VCO / PLL_P */
12 #define PLL_P      2
13
14 /* USB OTG FS, SDIO and RNG Clock =  PLL_VCO / PLLQ */
15 #define PLL_Q      6
```

Código 4.1: Modificaciones practicadas a system_stm32xx.c

La fuente de reloj es *HSE*, no se modifica —línea1—. El divisor *M* se pone a 8 —línea 7—, el multiplicador *N* a 288 —línea 9— , el divisor *P* a 2 y el divisor del reloj para el *USB OTG FS —PLL48CK—*, también controlado en el PLL principal, a 6. Con estos valores, la función *SystemInit* —perteneciente a este archivo—, cuando es llamada en el proceso de inicialización, configura *SYSCLK/HCLK* a 144 MHz, *PLL48CK* a 48 MHz y *PCLK2* 72 MHz, valor este último que asegura la máxima velocidad de muestreo en los *ADCs*. El archivo *system_stm32xx.c* dispone también, de una variable global llamada *SystemCoreClock*, encargada de contener el valor de la frecuencia del reloj del sistema —SYSCLK/HCLK— expresado en hercios, y que puede ser usada para configurar el temporizador interno el micro *Systick*. Por ello, su valor debe ser puesto al correcto:

```
1  uint32_t SystemCoreClock = 144000000;
```

Código 4.2: Variable SystemCoreClock del archivo system_stm32f4xx.c

4.3 Señales de prueba —archivo inicializa_hard—

Las señales para comprobar el funcionamiento del sistema, van a consistir en una señal cuadrada de 1 kHz —señal típica en un osciloscopio—, generada mediante el temporizador *TIM2*, accedida a través del pin 12 del puerto D, y tres parejas de señales que se generarán en los dos *DACs* —con ayuda del *DMA1*—, y accedidas a través de las salidas de los *DACs* PA.4 —pin 4 del puerto A, para *DAC1*— y PA.5 —pin 5 del puerto A, para *DAC2*—. La conmutación entre pares de señales se realizará mediante pulsaciones sucesivas en el pulsador de usuario —botón azul— de la palca *32F429IDISCOVERY*. Toda la configuración reside en el archivo *inicializa_hard.c*.

Para la configuración de la señal de 1 kHz se utiliza la función *configuraPrueba*, función que es llamada desde la función *main*, después de haber inicializado el librería *TouchGFX* —ver sección 3.5.2—, y la interrupción del temporizador 2 —TIM2—. La estructura básica del la función *configuraPrueba*, es la siguiente:

```
1  void configuraPrueba(void)
2  {
3   /* Declaración de las estructuras de inicialización */
4   TIM_TimeBaseInitTypeDef BaseTiempos;
5   NVIC_InitTypeDef NvicBaseTiempos;
6   GPIO_InitTypeDef  GPIO_InitStructure;
7
8   /* Habilitación de relojes de periféricos */
9   . . .
10
11  /* Configuración de TIM2 y su interrupción */
12  TIM_TimeBaseStructInit(&BaseTiempos);
13  BaseTiempos.TIM_Period=5-1;
14  BaseTiempos.TIM_Prescaler=7200-1;
15  TIM_TimeBaseInit(TIM2, &BaseTiempos);
16  TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE);
17  TIM_Cmd(TIM2, ENABLE);
18
19  NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
20  NvicBaseTiempos.NVIC_IRQChannel=TIM2_IRQn;
21  NvicBaseTiempos.NVIC_IRQChannelPreemptionPriority=7;
22  NvicBaseTiempos.NVIC_IRQChannelSubPriority = 0;
23  NvicBaseTiempos.NVIC_IRQChannelCmd=ENABLE;
24  NVIC_Init(&NvicBaseTiempos);
25
26  /* Configuración del pin GPIO_Pin_12 como salida en modo pushpull */
27  . . .
28 }
```

Código 4.3: Función para la configuración de la señal de prueba

Primeramente se declaran las estructuras de configuración del temporizador—TIM2— que generará la señal de prueba, del controlador de interrupciones anidadas —NVIC—, y del puerto —GPIOD—, por donde saldrá la señal de prueba—de forma respectiva, declaraciones en la líneas 4, 5 y 6—. A continuación se habilitan los relojes de estos periféricos —no mostrado en la porción de código—. Luego se configura el temporizador en sí, conociendo que obtiene su reloj del bus APB1 —PCLK1—, y este es de 36 MHz. Sin embargo, los relojes que llegan a los periféricos de tipo temporizador —en el micro STM32F429 y otros micros del fabricante ST— están multiplicados por dos, siendo, para este caso concreto, de 72 MHz el reloj que disciplina aTIM2. Para configurar el periférico como una temporización de forma repetida —al acabar de temporizar, se inicia de forma automática una nueva temporización—, se ha de especificar que el temporizador ha de trabajar en modo «base de tiempos» —línea 12—. La frecuencia se determina configurando los parámetros *Periodo* —*TIM_Period*, en línea 13— y

Prescalador —*TIM_Prescaler*, en línea 14—, especificando estos valores incrementados en una unidad. La expresión para calcular la frecuencia que generará el temporizador, es la siguiente [4]:

$$Frec = \frac{TIMCLK1}{2 \times \left((TIM_{Period} + 1) \times (TIM_{Prescaler} + 1)\right)} \quad (4.1)$$

El factor 2 que aparece en el denominador es debido a que la salida conmuta en cada periodo de temporización, por tanto, la frecuencia de salida es la mitad de la frecuencia correspondiente al periodo de temporización. En este caso y con los valores facilitados al temporizador, la frecuencia es de 1kHz. Para que se produzca la interrupción en cada periodo de temporización, se configura el periférico *TIM2* para ello —línea 16— y el *NVIC* habilita la interrupción *TIM2_IRQ* —línea 20—, con una prioridad de valor 7 —línea 21—. La rutina de atención a la interrupción es la siguiente:

```
1  /* Subrutina de atención a la interrupción de TIM2 (señal de prueba)*/
2  void TIM2_IRQHandler (void)
3  {
4      TIM_ClearITPendingBit(TIM2,TIM_IT_Update);//se borra la causa de interrupción.
5      GPIO_ToggleBits(GPIOD, GPIO_Pin_12);//se conmuta el estado del pin PD.12
6  }
```

Código 4.4: Rutina de atención a la interrupción para la señal de prueba

Lo que realiza, básicamente, es conmutar el pin 12 del puerto D en cada interrupción —cada actualización—.

Para las señales del generador, que irán conectadas a los dos canales de sistema, se definen los siguientes arrays que constituyen los datos para generar estas señales:

```
1  /*********************************************/
2  /*******    Definición de señales  ************/
3  /*********************************************/
4  //Primer par de señales (Seno12bit->Dac2, Escalera8bit->Dac1)
5  const uint16_t Seno12bit[32] = {
6      2047, 2447, 2831, 3185, 3498, 3750, 3939, 4056, 4095, 4056,
7      3939, 3750, 3495, 3185, 2831, 2447, 2047, 1647, 1263, 909,
8      599, 344, 155, 38, 0, 38, 155, 344, 599, 909, 1263, 1647};
9  const uint8_t Escalera8bit[6] = {0x0, 0x33, 0x66, 0x99, 0xCC, 0xFF};
10 //Segundo par de señales (Triangular12bit->Dac2, Cuadrada8bit->Dac1)
11 const uint16_t Triangular12bit[12] = {
12     2048,2730,3412,4095,3412,2730,2048,1366,684,0,684,1366};
13 const uint8_t Cuadrada8bit[8] = {255, 255, 255, 255, 0, 0, 0, 0};
14 //Tercer par de señales (SenoRectificado12bit->Dac2, Conmutacion8bit->Dac1)
15 const uint16_t SenoRectificado12bit[32] = {
16     2047, 2447, 2831, 3185, 3498, 3750, 3939, 4056, 4095, 4056,
17     3939, 3750, 3495, 3185, 2831, 2447, 2047, 2447, 2831, 3185,
18     3498, 3750, 3939, 4056, 4095, 4056, 3939, 3750, 3495, 3185, 2831, 2447};
19 const uint8_t Conmutacion8bit[8] = {0x80, 0xFF, 0xFF, 0xFF, 0x80, 0x00, 0x00, 0x00};
```

Código 4.5: Definición de las señales para el generador

Se definen en total 6 arrays que serán convertidos a través de los dos DACs en parejas de forma simultánea. Los nombres de cada array hacen referencia a la forma de onda que representan y al número de bits que se utilizarán en la conversión. El primer array de la pareja es convertido por DAC2, a 12 bits, mientras que el

segundo lo es por DAC1 a 8 bits. Se han utilizado pocos puntos en cada señal para evitar ocupar memoria RAM y para obtener una frecuencia relativamente alta en cada señal.

En la configuración del generador se utilizan los dos *DACs* que convierten cada par de señales, el *DMA1* con los flujos 5 y 6 que llevan cada uno de los datos de los arrays a los *DACs*, el temporizador *TIM6*, que disciplina los *DACs*, para solicitar una nueva petición *DMA*, y *GPIOA* para configurar los pines PA.4 y PA.5 como salidas de *DAC1* y *DAC2*, respectivamente. Esta configuración se hace dentro de la función *configuraGenerador* del archivo *inicializa_hard.c*, y es llamada desde la función *main* después de la inicialización de *TouchGFX* —ver sección 3.5.2—. La estructura de esta función es:

```
void configuraGenerador(void)
{
    /* Declaración de las estructuras de inicialización */
    . . .
    /* Habilitación de relojes de periféricos */
    . . .
    /* Configuración Canales del DAC 1 & 2 (DAC_OUT1 = PA.4)(DAC_OUT2 = PA.5) */
    . . .
    /* Configuración de la base de tiempos (TIM6) para los DACs
      (TIM6 está en APB1-> 36MHz x 2 = 72MHz*/
    . . .
    /* Configuración del canal 2 del DAC */
    DAC_StructInit(&DAC_InitStructure);
    DAC_InitStructure.DAC_Trigger = DAC_Trigger_T6_TRGO;
    DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;
    DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Enable;
    DAC_Init(DAC_Channel_2, &DAC_InitStructure);

    /* Configuración DMA1_Stream6 canal7 */
    DMA_DeInit(DMA1_Stream6);
    DMA_StructInit(&DMA_InitStructure);
    DMA_InitStructure.DMA_Channel = DMA_Channel_7;
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) &(DAC->DHR12R2);
    DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)&Triangular12bit;
    DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral;
    DMA_InitStructure.DMA_BufferSize = 32;
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
    DMA_Init(DMA1_Stream6, &DMA_InitStructure);
    DMA_Cmd(DMA1_Stream6, ENABLE);

    /* Se habilita el DAC2 */
    DAC_Cmd(DAC_Channel_2, ENABLE);

    /* Se Habilita el DAC2 para trabajar con el DMA */
    DAC_DMACmd(DAC_Channel_2, ENABLE);

    /* Configuración del canal 1 del DAC */
    . . .
    /* Configuración DMA1_Stream5 canal7 */
    . . .
    /* Se habilita el botón de usuario (botón azul de la palca)
             lanzar la interrupción EXTI0_IRQ                */
    STM_EVAL_PBInit(BUTTON_USER, BUTTON_MODE_EXTI);
}
```

Código 4.6: Función para la configuración del generador

Al igual que para la función *configuraPrueba*, primeramente se declaran las estructuras de inicialización para *TIM6*, *GPIOA*, *DAC* y *DMA1*, así como la habilitación de sus relojes. Luego, se configura *GPIOA* para que sus pines PA.4 y PA.5 sean las salidas respectivas de *DAC1* y *DAC2* en modo analógico sin resistencia de pull. A continuación, se configura *TIM6* de forma análoga a como se hizo para *TIM2*, estableciendo *TIM_Period* a 100 y *TIM_Prescaler* a cero. Con estos valores, y ya que *TIM6* está igualmente en el bus *APB1*, su periodo de temporización es de, aproximadamente, 1,40278 microsegundos. Esto hace que las señales de 32 datos tengan una frecuencia de 22,28 kHz, y las de 6 datos, de 118,81 kHz. Una configuración particular practicada a *TIM6* —no mostrada en la porción de código—, es la conexión del evento de actualización —cada temporización— a su salida interna para poder disciplinar los *DACs*. A continuación, se configura el *DAC2*, especificando que estará disciplinado por *TIM6* —línea 13—, que no se genera internamente ninguna señal —línea 15— y que se habilita su amplificador de salida —línea 16—. Para posibilitar el paso de los datos del *array* correspondiente hacia el *DAC2*, si la intervención del núcleo del micros, se utiliza el periférico *DMA1*, flujo 6, canal 7. En él se especifica que la transferencia de datos será de memoria hacia periférico —línea 25—, el *array* inicial a transferir será *triangular12bits* —línea 24—, y el destino, el registro que el DAC2 utiliza para convertir —línea 23—, que el número de datos a transferir es de 32, el tamaño del *array* —línea 26—, que cada vez que se transfiere un dato, la dirección de memoria se incrementa pero la del periférico no —líneas 28 y 27, respectivamente—, que el tamaño del dato a transferir de la memoria, y se recibe en el periférico es de 16 bits —líneas 30 y 29, respectivamente—, que cuando finalice la transferencia del *array* se volverá a reiniciar la transferencia —modo circular, línea 31—, que no se utilizará la FIFO del *DMA* —línea 33— y que la prioridad de la transferencia, frente a otras fuentes que utilicen el *DMA1*, es alta —línea 32—. Para finalizar con la configuración del *DAC2*, se procede a su habilitación —línea 41—, a la habilitación del *DMA1* flujo 6 —línea 38— y a la habilitación de la petición *DMA* por parte del *DAC2* —línea 44—. Para el caso de *DAC1* existe una configuración similar, con las principales deferencias que utiliza el flujo 5 del *DMA1*, que el *array* inicial a transferir es *Cuadrada8bit*, que el número de datos a transferir es de 6 y el tamaño de datos es de 8 bits. La configuración final que realiza esta función, es establecer la interrupción EXTI0 para el pulsador azul de la palca DISCO a través de la función de la librería de periféricos de la placa —línea 52—. Esta última función configura el NVIC para la interrupción EXTI0 para una prioridad de 15. Esta prioridad no es muy preferente, pero no la necesita ya que se trata de una interrupción desde un pulsador.

Para poder cambiar los pares de señales del generador, se define un tipo enumerado con tres valores correspondientes a los tres estados posibles del generador —los tres pares de señales—, así como la declaración de una variable —estática— que este tipo, que mantenga el esta actual:

```
1  typedef enum
2  {
3      UNO,
4      DOS,
5      TRES
6  }TEstado_generador;
7
8  static volatile TEstado_generador estado_generador=DOS;
```

Código 4.7: Control de estados del generador

Entre las líneas 1 a 6, está la definición del tipo «TEstado_generador» que establece los tres estados en los que se puede encontrar el generador. En la línea 8 se declara la variable «estado_generador» del tipo anterior para mantener el estado actual. Esta variable es de tipo volatile ya que va ha ser utilizada por una interrupción y no se desea que el compilador realice optimizaciones sobre la misma.

La rutina de interrupción que se ejecutará al accionar el pulsador azul de la palca es:

```
1  void EXTI0_IRQHandler(void)
2  {
3      if (EXTI_GetITStatus(USER_BUTTON_EXTI_LINE) != RESET)
4      {
5          DMA_Cmd(DMA1_Stream6, DISABLE);
6          DMA_Cmd(DMA1_Stream5, DISABLE);
7
8          switch (estado_generador)
9          {
10         case UNO:
11             DMA1_Stream6->M0AR=(uint32_t)Triangular12bit;
12             DMA_SetCurrDataCounter(DMA1_Stream6, 12);
13
14             DMA1_Stream5->M0AR=(uint32_t)Cuadrada8bit;
15             DMA_SetCurrDataCounter(DMA1_Stream5, 8);
16
17             estado_generador=DOS;
18             break;
19         case DOS:
20             DMA1_Stream6->M0AR=(uint32_t)SenoRectificado12bit;
21             DMA_SetCurrDataCounter(DMA1_Stream6, 32);
22
23             DMA1_Stream5->M0AR=(uint32_t)Conmutacion8bit;
24             DMA_SetCurrDataCounter(DMA1_Stream5, 8);
25
26             estado_generador=TRES;
27             break;
28         case TRES:
29             DMA1_Stream6->M0AR=(uint32_t)Seno12bit;
30             DMA_SetCurrDataCounter(DMA1_Stream6, 32);
31
32             DMA1_Stream5->M0AR=(uint32_t)Escalera8bit;
33             DMA_SetCurrDataCounter(DMA1_Stream5, 6);
34
35             estado_generador=UNO;
36             break;
37         }
38         DMA_Cmd(DMA1_Stream6, ENABLE);
39         DMA_Cmd(DMA1_Stream5, ENABLE);
40
41         EXTI_ClearITPendingBit(USER_BUTTON_EXTI_LINE);
42     }
43 }
```

Código 4.8: Interrupción que controla es estado del generador

Al pulsar sobre el botón azul y ejecutar esta rutina de interrupción, lo primero que se hace es inhabilitar las transferencias *DMA* —líneas 5 y 6—, para a continuación, partiendo del estado que indica la variable «estado_generador», determinar el siguiente estado al que ir, y con él, la señales a sacar por los dos *DACs* —líneas de la 8 a la 27. Luego se habilitan nuevamente las transferencias DMA —líneas 28 y 29—.

4.4 Componentes gráficos específicos para el proyecto

En esta sección se muestran los componentes gráficos que han sido utilizados en el proyecto pero que no vienen incluidos en la librería de TouchGFX. Estos componentes son, tanto de nueva creación, como adquiridos de los códigos ejemplo, que vienen incluidos en la distribución, o en el repositorio o foro de la página web del fabricante. Así, esta sección se divide en aquellos nuevos componentes que son de creación propia, y aquellos que, no perteneciendo a la librería, son adquiridos del fabricante.

4.4.1 Componentes creados

4.4.1.1 Componente gráfico «Pulsador»

Se trata realmente de una especialización de controles de tipo *Button* o derivados de este —más concretamente derivados desde la clase abstracta *AbstractButton*—. Por ello, es una clase plantilla cuyo tipo indeterminado es el tipo concreto, descendiente de *AbstractButton*, de la clase a la que se quiere aplicar la característica de pulsador. Su pretensión es dar la habilidad, a la clase a la que especializa, de poder gestionar el evento de pulsación —que se dispara nada más tocar el componente—, en contraposición al evento *Click*, que se dispara al pulsar sobre el componente seguido de la liberación sin salirse de los límites del mismo —que también se ha de gestionar con esta plantilla—. El tipo plantilla *TPulsador*, es creado para este proyecto para dar esta característica a los botones de ocultación —ver secciones 4.4.1.6 y 4.4.1.7— que aparecen en pantalla al tocar la gráfica, y que sirven para modificar la escala horizontal y vertical del mismo. Un código similar a esta plantilla fue mostrado en la sección que se explicaba cómo crear un componente nuevo a partir de la especialización de uno ya existente —sección 1.13.1—, pero en el caso del componente mostrado aquí, se hace necesario que se trate de una plantilla, ya que se pretende dar esta característica a otras clase diferentes de *Button*, aunque sí en la misma rama de herencia. Su código es el siguiente:

```
1  template<class T>
2   class TPulsador : public T
3   {
4    public:
5    TPulsador():T(), estado(TPulsador::LIBERADO)
6    {}
7    typedef enum
8    {
9      PULSADO,
10     LIBERADO
11   }Estados;
12   virtual void handleClickEvent (const ClickEvent &event)
13   {
14    T::handleClickEvent(event);
15    if(event.getType()==ClickEvent::PRESSED && estado==LIBERADO)
16    {
17     estado=PULSADO;
18      if (EventoPulsadoCallback && EventoPulsadoCallback->isValid())
19       {
20        EventoPulsadoCallback->execute(*this);
21       }
22     }
23    else
24     {
25      estado=LIBERADO;
26      if (EventoPulsadoCallback && EventoPulsadoCallback->isValid())
27       {
28        EventoPulsadoCallback->execute(*this);
29       }
30     }
31   }
32
33   bool getState(void) const{return(estado==PULSADO)? true : false;}
34   void setAction(GenericCallback<const AbstractButton& >\
35                  &retrollamada){EventoPulsadoCallback=&retrollamada;}
36   protected:
37    Estados estado;
38    GenericCallback<const AbstractButton& >*EventoPulsadoCallback;
39  };
40  typedef TPulsador<Button> Pulsador;
```

Código 4.9: Código de la plantilla TPulsador

El tipo indeterminado se denota como T en la especificación de plantilla —línea 1— así como dentro de la clase en diferentes partes de su código —líneas 2, 3 y 14—. Para el manejo interno, se define el tipo *Estados* —de tipo enumeración— con los posibles valores PULSADO Y LIBERADO, y su correspondiente atributo privado, *estado*, que indicará, en cada momento, si el botón ha sido pulsado o liberado. Como atributo, también se define

un puntero a objeto de tipo *GenericCallback*, llamado *EventoPulsadoCallback*, con el fin de llamar al evento de usuario al ser pulsado o liberado el objeto de esta clase. La determinación de si se ha pulsado sobre el control, se realiza sobre la rescritura del método *handleClickEvent* —líneas de la 12 a la 31—, donde la primera acción a llevar a cabo es ceder la manipulación de este evento a la clase antecesora —línea 14—, con el fin de no modificar el comportamiento que ya tenía la clase —a la que se le aplica esta plantilla— cuando manipulaba el evento *Click*. Luego, se procede a determinar si el objeto ha sido pulsado —líneas de la 15 a la 21—, utilizando para ello la información de su estado, que viene en el objeto del evento —línea 15—. Si es así, el control ha sido pulsado y se procede a llamar al evento de usuario —si existe—, con la asignación previa del atributo *estado* a PULSADO. Si no se ha producido la pulsación es que se ha producido la liberación —ya que el evento de pulsación ha debido suceder antes—, e igualmente, se llama al manipulador de usuario previa asignación del atributo *estado* a LIBERADO. Este atributo interno —*estado*—, es posible acceder a él desde fuera de la clase a través del método *getState* —línea 33—. Esto permite dos cosas. La primera de ellas, discernir si se ha producido el evento por pulsación o liberación, y la segunda, permitir que sea el mismo manipulador para gestionar el evento *Click* —pulsación seguida de liberación—, el que gestione también el evento de pulsación y liberación por separado. De esta forma el usuario solo necesita un manipulador para gestionar la pulsación, la liberación o la pulsación seguida de liberación —*Clikc*—, consultando para ello, al método *getState*. Para aclarar su utilización, se presenta la declaración de un objeto de tipo botón al que se la aplica esta plantilla —en la parte privada de la vista— para poder gestionar, además del evento *Click*, la pulsación y liberación por separado:

```
1  class MiVista : public View<MiPresentador>
2  {
3   public:
4    MiVista():
5         AlBotonValorCallback(this, &MiVista::Al_Boton)
6              {}
7    . . .
8    void Al_Boton(const AbstractButton &boton);
9   private:
10   TPulsador<Button> MiPulsador;
11   Callback<MiVista, const AbstractButton &> AlBotonValorCallback;
12 };
```

Código 4.10: Declaración del objeto de tipo TPulsador

Además de la declaración del objeto tipo *TPulsador* —*MiPulsador* en línea 10—, se declara un objeto de tipo *Callback* para que pueda llamar al evento de usuario en la pulsación y liberación —línea 11—, y la declaración de la función que actuará de manipulador —línea 8—. La definición de esta función se muestra a continuación:

```
1  void MiVista::Al_Boton(const AbstractButton &boton)
2  {
3    if(&boton==&MiPulsador)
4    {
5     if(MiPulsador.GetState()==TPulsador::PULSADO)
6              //realizar la acción al pulsar
7     else if (MiPulsador.GetState()==TPulsador::LIBERADO)
8              //realizar la acción al liberar
9    }
10 }
```

Código 4.11: Manipulador de evento de la clase TPulsador

Inicialmente se verifica que la causa del evento esté originada desde el objeto *MiPulsador* —línea 3—. Luego, mediante el método *GetState*, se discrimina si el evento es de pulsación —línea 5— o liberación —línea 7—.

4.4.1.2 Componente gráfico «TouchAreaMio»

La pretensión de este componente es poder gestionar eventos de arrastre por el gráfico y eventos de cancelación de pulsación, que no son tratados por el control de que desciende —*TouchArea*— y que pertenece a la librería de componentes suministrados con *TouchGFX*. Se trata, pues, de la creación de un nuevo componente mediante especialización de uno ya existente —ver sección 1.13.1—. La necesidad de tratar el evento de cancelación es debido a que sobre el gráfico aparecerán botones de cambio de escala vertical y horizontal y cuando se produzca la cancelación de estos botones —pulsar sobre ellos y sin liberar la pulsación se arrastre hasta una zona fuera de sus límites y la consecuente liberación—, han de iniciar su desaparición:

Ilustración 4.13: Control TouchAreaMio y botones de ocultación controlados por él

En el caso del evento arrastre, a pesar de estar definido en la clase antecesora, *TouchArea*, se hace de forma virtual y con cuerpo vacio de la función, con lo que es necesario redefinirlo en la clase descendiente, es decir, en esta —*TouchAreaMio*—, dando el código que dé respuesta al evento. La clase base, *TouchArea*, implementa un mecanismo para detectar la presión sobre el área —además de detectar la presión seguida de liberación, lo que es conocido como evento *Click*—. Esto se consigue mediante un puntero de tipo *GenericCallback*, con la misma firma que el utilizado para el evento *Click* —esta plantilla solo tiene como tipo indeterminado el objeto que dispara el evento, lo que posibilita utilizar el mismo manipulador de usuario del evento Click—, un método que carga este puntero, y el método de manipulación de evento de componente *handleClickEvent*. Hay que destacar que este último manipulador de evento, sí tiene implementación en la clase base —*TouchArea*—, con lo que para gestionar solo la presión, habrá que detectar que el disparo del mismo ha sido solo por presión —y no por presión seguida de liberación—, dentro del evento de componente *handleClickEvent*. En definitiva, se ha de gestionar el evento de arrastre, dentro del método de manipulación de evento de arrastre de usuario—llamado desde *handleDragEvent*—, y los eventos de presión, *click* y cancelación, dentro del evento *click* de usuario —llamados desde *handleClickEvent*—.

El código de este nuevo componente creado, está implementado, exclusivamente, en un archivo de cabecera —*TouchAreaMio.hpp*—, y es el siguiente:

```
1  #ifndef TOCHCAREA_MIO
2  #define TOCHCAREA_MIO
3
4  #include <touchgfx/widgets/TouchArea.hpp>
5  #include <touchgfx/Callback.hpp>
6  class TouchAreaMio : public touchgfx::TouchArea
7  {
8      public:
9          virtual void handleClickEvent(const touchgfx::ClickEvent& event)
10         {
11             TouchArea::handleClickEvent(event);
12             if (event.getType() == touchgfx::ClickEvent::CANCEL)
13             {
14                 if (Cancelacion && Cancelacion->isValid())
15                 {
16                     Cancelacion->execute(*this);
17                 }
18
19             }
20         }
21
22         virtual void handleDragEvent(const DragEvent& evt)
23         {
24             TouchArea::handleDragEvent(evt);
25             if (Arrastre && Arrastre->isValid())
26             {
27                 Arrastre->execute(*this, evt);
28             }
29         }
30
31
32         void setCancelacion(GenericCallback< const AbstractButton & >& callback)
33         {
34             Cancelacion = &callback;
35         }
36
37         void setArrastre(GenericCallback< const AbstractButton &,
38                          const DragEvent&>& callback)
39         {
40             Arrastre = &callback;
41         }
42
43     private:
44         GenericCallback< const AbstractButton &  > *Cancelacion;
45         GenericCallback< const AbstractButton &, const DragEvent&> *Arrastre;
46 };
47 #endif
```

Código 4.12: Control TouchAreaMio

Se redefine el manipulador de evento *Click* de componente —líneas de la 9 a 20—, con el fin de detectar la cancelación sobre el área y disparar el consecuente evento de usuario. Esto se realiza en el bloque condicional entre las líneas 12 y 19. Primeramente, en el manipulador *handleClickEvent*, se ejecuta el manipulador homónimo de la clase base —línea 11—, esto hace que se dispare el evento *Click* de usuario si ha sido establecido, luego, en la condición de la línea 12, se comprueba que la causa ha sido por cancelación, y si es así, se desreferencia el puntero de tipo *GenericCallback* —*Cancelacion*, declarado en la línea 44—, ejecutando el manipulador de evento para la cancelación del usuario. Entre la líneas 22 a 29 se define el manipulador de evento *Drag* de componente —*handleDragEvent*—. Este evento, como ya ha sido mencionado, tiene definición nula en la clase antecesora —*TouchArea*—. A pesar de ello, dentro de esta clase —*TouchAreaMio*—, se llama al método homónimo de la clase antecesora —línea 24—. Luego, se comprueba que existe un manipulador de usuario, y si es así, se llama —línea 27—. Los punteros respectivos a los manipuladores de usuario —de tipo *GenericCallback*—, están declarados en las líneas 44 y 45.

El código de este componente, en la cabecera de la vista es:

```
1  class FftView : public View<FftPresenter>
2  {
3  public:
4     FftView():
5               BotonCallback(this, &FftView::Al_Boton),
6               TouchAreaArrastradaCallback(this, &FftView::Al_TouchAreaArrastrada)
7
8     {}
9     . . .
10    void Al_Boton(const AbstractButton &button);
11    void Al_TouchAreaArrastrada(const AbstractButton &area, const DragEvent &evento);
12
13 private:
14    . . .
15
16    TouchAreaMio areaTocable;
17    . . .
18    Callback<FftView, const AbstractButton &> BotonCallback;
19    . . .
20    Callback<FftView, const AbstractButton &, const DragEvent&> TouchAreaArrastradaCallback;
21 };
```

Código 4.13: Declaración del control de tipo TouchAreaMio

En la parte privada se declaran los objetos de tipo *Callback* que posibilitan la llamada a eventos de usuario cuando son cargados con los puntero del objeto y método de este, que actúa como manipulador de evento de usuario —líneas 5 y 6—, y son cargados en los punteros internos de *TouchAreaMio* de tipo *GenericCallback*. En las líneas 10 y 11 están las declaraciones de los manipuladores de usuario para manejar los eventos de pulsación, *click* y cancelación —método *Al_Boton*— y el evento de arrastre —método *Al_TouchAreaArrastrable*—. Por supuesto, se ha de crear un objeto de tipo *TouchAreaMio* —*areaTocable* en la línea 16—. Para configurar este objeto, en el método *setupSecreen* de la vista, se localiza el siguiente código:

```
1  areaTocable.setPosition(0,0,480,181);
2  areaTocable.setPressedAction (BotonCallback);
3  areaTocable.setAction(BotonCallback);
4  areaTocable.setCancelacion(BotonCallback);
5  areaTocable.setArrastre(TouchAreaArrastradaCallback);
```

Código 4.14: Configuración del control de tipo TouchAreaMio

El objeto va a ocupar toda el área de la gráfica —línea 1— y se ha de posicionar encima de esta, es decir ha de ser añadido a la vista después de que se añada la gráfica. Luego se especifica cuáles serán los manipuladores de evento de usuario —líneas 2, 3 y 4— para los eventos respectivos de presión, *click* y cancelación —manejados por el manipulador de usuario *Al_Boton*, llamado desde el manipulador de componente *handleClickEvent*—, y que el manipulador de usuario para el evento de arrastre será el método de la vista *Al_TouchAreaArrastrable* —línea 5—.

La definición del manipulador de evento de usuario *Al_Boton* es la siguiente:

```
1  void FftView::Al_Boton(const AbstractButton &button)
2  {
3   if(&button==&areaTocable)
4    {
5     if(areaTocable.getPressedState())
6      {
7        grupoOcultadoV.iniciaAparicion();
8        grupoOcultadoH.iniciaAparicion();
9      }
10    else
11     {
12       grupoOcultadoV.iniciaOcultacion();
13       grupoOcultadoH.iniciaOcultacion();
14     }
15   }
16 }
```

Código 4.15: Manipular Al_Boton del control del tipo TouchAreaMio en el proyecto

Como este manipulador de usuario va a compartir código procedente de varios controles, inicialmente, es necesario identificar cual es el que ha disparado el evento, en este caso, se trata del objeto *areaTocable* —línea 3—. Luego, hay que discernir si el evento se ha producido por presión o por *click* y cancelación. Si es por presión —línea 5—, se procede a visualizar los botones de ajuste vertical —agrupados en el objeto *grupoOcultadoV*, ver secciones 4.4.1.6 y 4.4.1.7— y horizontal — agrupados en el objeto *grupoOcultadoH*—; si es por *click* o cancelación —línea 10—, se procede a ocultarlos.

Finalmente, la definición del manipulador del evento de arrastre de usuario, es la siguiente:

```
1  void FftView::Al_TouchAreaArrastrada(const AbstractButton &area, const DragEvent &evento)
2  {
3    if(&area == &areaTocable)
4     {
5       if(serieFFT.obtenEscalaHor()>=1)
6          serieFFT.ponPosicionHor(serieFFT.obtenPosicionHor()-\
7                                  evento.getDeltaX()*serieFFT.obtenEscalaHor());
8       else
9          serieFFT.ponPosicionHor(serieFFT.obtenPosicionHor()- evento.getDeltaX());
10     presenter->obtenerModelo()->ponerDespHor(serieFFT.obtenPosicionHor());
11     actualizaTextoCursores();
12    }
13 }
```

Código 4.16: Manipular Al_TouchAreaArrastrable del control del tipo TouchAreaMio en el proyecto

Igualmente que en el caso anterior, se ha identificar el objeto que produce el evento, ya que es posible que otro objeto también utilice este mismo manipulador —línea 3—. Este tipo de evento, además de tener como argumento el objeto que lo dispara, tiene un segundo, de tipo *DragEvent*, que se trata de un objeto que, entre otros métodos, dispone de uno que informa cuantos puntos se han producido al arrastrar horizontalmente por pantalla —método *getDeltaX*—. Para aplicar el desplazamiento en la gráfica, que corresponde al arrastre producido en pantalla, se ha de discernir, previamente, si la gráfica está dibujando varios datos sobre un mismo punto horizontal en pantalla o no —ver sección 4.4.1.11—. Si se están dibujando varios datos sobre la misma horizontal —condición en la línea 5—, se han de desplazar todos los datos correspondientes a esa misma horizontal de pantalla a la vez —línea 6 y 7—. Si no, cada dato individual sufrirá el desplazamiento correspondiente al arrastre horizontal —línea 9—. En cualquiera de los casos, se ha de actualizar cual es el

desplazamiento horizontal de los datos en pantalla que el modelo almacena y que corresponde con la posición horizontal de la gráfica —línea 10—. Esta actualización en el modelo provoca el disparo de un mensaje al presentador y este responde al mismo actualizando todos los textos referentes a frecuencia. También se procede a actualizar la lectura de los cursores —línea 11— si estos están visibles —ver sección 4.4.1.12—. El incremento se resta al desplazamiento actual —líneas 6-7 y 9-10— porque cuando se hace un arrastre hacia la izquierda —incremento negativo, dado por *getDeltaX*—, se pretende aumentar la posición en la visualización de la gráfica —desplazarla hacia la izquierda—, y cuando se produce un arrastre hacia la derecha —incremento positivo, dado por *getDeltaX*—, se pretende disminuir la posición en la visualización de la gráfica, es decir, desplazarla hacia la derecha.

4.4.1.3 Componente gráfico «Marcador»

Este es un componente cuya única misión va a ser la de indicar, de forma gráfica, el acontecer de un suceso. Esto lo hará en forma de parpadeo, utilizando la transparencia alfa para desvanecerse, en un periodo de tiempo fijado, para a continuación aparecer, con otro tiempo programado. El inicio de este parpadeo lo ha de llevar a cabo algún otro componente que quiera indicar un suceso, u otra parte del código, llamando a su método *iniciar*. Esto significa que no puede gestionar ningún evento de la capa HAL a no ser el de temporización, aunque sí puede disparar un evento de usuario, de forma continúa en cada *tick* del entorno gráfico, cuando se encuentra en su estado de parpadeo. Aún así, siempre se puede usar una plantilla de tipo *mixin* para que sí pueda aceptar ciertos eventos. Su aspecto es el siguiente:

Ilustración 4.14: Apariencia del control Marcador en este proyecto

Se trata de un componente que solo dispone de archivo de cabecera —Marcador.hpp—, ya que, debido a su simplicidad, se decide incluir todo su código en este archivo. La parte privada de la clase es la siguiente:

```
1   class Marcador : public Image
2   {
3       public:
4           Marcador() :
5                   Image(),
6                   Ecuacion(&EasingEquations::linearEaseNone),
7                   duracion_ascendente(15),
8                   duracion_descendente(0),
9                   contador_ascendente(0),
10                  contador_descendente(0),
11                  alfaMinimo(100),
12                  alfaMaximo(200),
13                  iniciado(false),
14                  vueltas(0)
15          {
16          }
17          . . .
18      private:
19          EasingEquation Ecuacion;
20          uint16_t  duracion_ascendente;
21          uint16_t  duracion_descendente;
22          uint16_t  contador_ascendente;
23          uint16_t  contador_descendente;
24          uint8_t   alfaMinimo;
25          uint8_t   alfaMaximo;
26          bool      iniciado;
27          uint32_t  vueltas;
28          GenericCallback<const Marcador&>* GenericCallbackMarcando;
29
30          virtual void handleTickEvent()
31          {
32            . . .
33          }
34  };
```

Código 4.17: Definición de la clase Marcador

Se trata de la creación de una componente por especialización de uno ya existente —Image—, es por ello que posee todos los atributos y método de la clase heredada. Dispone de una ecuación —*Ecuacion*, en línea 19— que va a controlar tanto el desvanecimiento como la aparición. Los tiempos respectivos expresados en ticks—, están fijados por los atributos *duracion_descendente* y *duracion_ascendente* —líneas 20 y 21—. La cuenta de los pasos que se llevan a cabo hasta llegar a estos dos valores, están almacenados en los atributos respectivos *contador_descendente* y *contador_ascendente* —líneas 22 y 23—. El valor de transparencia al que se llega tras la aparición es el fijado por *alfaMaximo*—línea 25—, mientras que al que se llega tras la desaparición es *alfaMinimo* —línea 24—. El atributo *iniciado* —línea 26—indica que la animación está en curso. Para poder llamar a un manipulador de usuario, mientras el objeto de tipo *Marcador* está parpadeando, se declara el puntero *GenericCallbackMarcando* —línea 28—. En la línea 27, el atributo vueltas, fija los *ticks* que han de pasar para volver a llamar al manipulador de usuario desde el puntero anterior. En la parte privada se define el evento de temporización —línea 30— que gestionará el periodo de parpadeo así como la llamada al evento de usuario. En la parte pública de la clase están los métodos necesarios para poder cambiar cada uno de los atributos antes descritos —a excepción del atributo *Ecuacion* que se fija en el constructor a *linearEaseNone*—. Así mismo, el resto de atributos es inicializado en el constructor de la clase —línea 4—. Para iniciar y parar el parpadeo, la clase dispone de estos dos métodos:

```
1   void iniciar(void) {Application::getInstance()->registerTimerWidget(this); iniciado=true;}
2   void parar(void) {Application::getInstance()->unregisterTimerWidget(this); iniciado=false;}
```

Código 4.18: Inicio y detención de la temporización de la clase Marcador

El primer método —línea 1— registra el objeto en la instancia de la clase Application, para que pueda recibir el evento de temporización, y a continuación indica que la animación a comenzado. El segundo método —línea 2—, realiza la labor contraria, quita del registro el componente para no recibir el evento de temporización, y luego indica que la animación ha concluido. El corazón del comportamiento de este componente se encuentra en su manipulador de evento de componente *handleTickEvent*:

```
1  virtual void handleTickEvent()
2   {
3    static uint32_t contadorVueltas=0;
4    if (iniciado)
5     {
6      if (contador_ascendente <= (uint32_t) (duracion_ascendente))
7       {
8         uint8_t deltaAlpha = (uint8_t) Ecuacion(contador_ascendente, 0,\
9                               alfaMaximo - alfaMinimo, duracion_ascendente);
10        Image::setAlpha(alfaMinimo + deltaAlpha);
11        Image::invalidate();
12        contador_ascendente++;
13         if (contador_ascendente >= (uint32_t) (duracion_ascendente))contador_descendente=0;
14      }
15     else if (contador_descendente <= (uint32_t) (duracion_descendente))
16      {
17        uint8_t deltaAlpha = (uint8_t) Ecuacion(contador_descendente, 0,\
18                              alfaMaximo - alfaMinimo, duracion_descendente);
19        Image::setAlpha(alfaMaximo - deltaAlpha);
20        Image::invalidate();
21        contador_descendente++;
22        Image::invalidate();
23        if (contador_descendente >= (uint32_t) (duracion_descendente))contador_ascendente=0;
24      }
25     if (GenericCallbackMarcando && GenericCallbackMarcando->isValid() &&\
26                                        ((contadorVueltas % vueltas )==0))
27      {
28         GenericCallbackMarcando->execute(*this);
29      }
30     contadorVueltas ++;
31    }
32  }
```

Código 4.19: Definición del evento temporización de la clase Marcador

Solo se procede a ejecutar el código de este método si se ha iniciado —línea 4—. Este método está dividido en tres partes. La primera —líneas de la 6 a la 14—, se gestiona la aparición. Para ello cuenta con el contador de animación —*contador_ascendente*—, la ecuación que determina el incremento en cada paso, y el incremento total, que es el valor diferencia entre el mayor valor alfa y el menor. El incremento determinado en cada *tick*, es sumado al valor alfa actual hasta llegar al máximo. De igual forma se procede en la segunda parte de esta función, dedicada a la desaparición –líneas de la 15 a la 24—. La diferencia es que el incremento en cada paso es restado del valor actual alfa hasta llegar al mínimo. En cualquier caso, ya se esté en la fase de aparición o en la desaparición, en la tercera parte de esta función —línea 25—, se llama al manipulador de usuario si este existe. La cadencia de llamada dependerá del atributo vueltas. Este proceso de aparición y desaparición es perpetuo a no ser que se llame al método *parar*.

Como ejemplo de declaración de un objeto de este tipo, se muestra el utilizado en el proyecto para indicar que la adquisición está detenida y además, al aplicarle la platilla *mixin* de tipo *ClickListener*, es capaz de gestionar el evento *Click*:

```
1  class FftView : public View<FftPresenter>
2  {
3  public:
4   FftView():
5   . . .
6   MarcadorCallback(this, &FftView::Al_Marcador),
7   MarcadorActuandoCallback(this, &FftView::Al_MarcadorActuando),
8   . . .{}
9
10  void Al_Marcador(const Marcador& marcador, const ClickEvent& evento);
11  void Al_MarcadorActuando(const Marcador& marcador);
12
13 private:
14  . . .
15  ClickListener<Marcador> ledActivador;
16  Callback<FftView, const Marcador&, const ClickEvent&> MarcadorCallback;
17  Callback<FftView, const Marcador&> MarcadorActuandoCallback;
18  . . .
19 }
```

Código 4.20: Declaración del objeto de tipo Marcador en la cabecera de la vista

La declaración del objeto, modificado por la plantilla *mixin*, aparece en la línea 15. Esto le capacita para recibir el evento *CLick*, es decir para que pueda reaccionar al ser tocado. En las líneas 16 y 17 se declaran los objetos de tipo *Callback* que van a hacer de puente entre el objeto *ledActivador* —de tipo *Marcador* modificado— y los manipuladores de evento de usuario, declarados en las líneas 10, para gestionar el evento *Click*, y en la 11, que es llamado constantemente, en cada *tick*, mientras el objeto de tipo Marcador está animado —*parpadeando*—.

Los objetos de tipo *Callback* son inicializados en el constructor de la vista —líneas 6 y 7—. Como cualquier otro componente gráfico, su configuración se realiza en el método *setupScreen* de la vista:

```
1  void FftView::setupScreen()
2  {
3   . . .
4   ledActivador.setBitmap(Bitmap(BITMAP_LED_ID));
5   ledActivador.setXY(305,3);
6   ledActivador.ponerAlfaMaximo(250);
7   ledActivador.ponerAlfaMinimo(10);
8   ledActivador.ponerDuracionAscendente(18);
9   ledActivador.ponerDuracionDdescendente(18);
10  ledActivador.setClickAction(MarcadorCallback);
11  ledActivador.ponerAccionMarcando(MarcadorActuandoCallback);
12  . . .
13  add(ledActivador);
14  . . .
15 }
```

Código 4.21: Definición del objeto tipo Marcador en el archivo fuente de la vista

Inicialmente se configuran cuestiones de apariencia como son la imagen que va a tener y su posición —líneas 4 y 5, respectivamente—. Para ello, se utilizan métodos heredados de la clase *Image*. Luego, se configura la animación estableciendo los niveles de transparencia máximo y mínimo —líneas 6 y 7, respectivamente—, y los tiempos de duración de aparición y desvanecimiento —líneas 8 y 9, respectivamente—. A continuación, se le especifica al objeto qué manipulador de usuario se utilizará al ser tocado el componente —línea 10—, y cuál será el manipulador de usuario a ser llamado de forma constantemente durante la animación —línea 11—. Finalmente el objeto es añadido a la vista —línea 13—.

```
1  void FftView::Al_Marcador(const Marcador& marcador, const ClickEvent& evento)
2  {
3    if((&marcador == &ledActivador) && (evento.getType() ==  ClickEvent::RELEASED))
4     {
5      if(ledActivador.estaIniciado())
6       {
7        ledActivador.parar();
8        ledActivador.setAlpha(250);
9        presenter->obtenerModelo()->ponerAdquiere(true);
10      }
11     else
12      {
13       ledActivador.iniciar();
14       presenter->obtenerModelo()->ponerAdquiere(false);
15      }
16     actualizaTextoCursores();
17    }
18 }
```

Código 4.22: Manipulador de usuario del evento Click del objeto tipo Marcador

La pretensión de este manipulador es la de parar y activar la adquisición de datos, de forma alternada al ser tocado el objeto de tipo *Marcador*, y que este indique —marque— el estado de la adquisición —si está animado la adquisición está detenida, mientras que si no está animado, la adquisición está en marcha—. Como en cualquier manipulador, lo primero es discernir qué objeto ha producido el evento y en qué condiciones —línea 3—. Luego, concretamente para este manipulador, se discierne si el objeto de tipo *Marcador* —l*edActivador*— está animado —línea 5— o si no lo está —línea 11—. Si lo está, se para la animación, se pone la transparencia del objeto a casi opaca y se manda al modelo a que reanude la adquisición. Si no lo está, se inicia la animación y se instruye al modelo a detener la adquisición. En cualquier caso, tanto al detener como al activar la adquisición, se procede a actualizar los textos que indican las lecturas de los cursores —línea 16—.

El manipulador de usuario para el evento propio del objeto de tipo Marcador, para este ejemplo, es el siguiente:

```
1  void FftView::Al_MarcadorActuando(const Marcador& marcador)
2  {
3    if(&marcador == &ledActivador)
4     {
5      graphContainer.invalidate();
6     }
7  }
```

Código 4.23: Manipulador de usuario para el evento del objeto de tipo Marcador

Este manipulador tiene por objeto refrescar constantemente el gráfico cuando la adquisición está detenida. Esto permite escalar el gráfico horizontalmente o actualizar las lecturas de los cursores cuando estos son movidos por pantalla o la pantalla es movida a través de ellos, cuando la adquisición está detenida. Para conseguirlo, este manipulador manda redibujar el gráfico —línea 5—. Hay que recordar que este manipulador de evento se ejecuta en cada *tick* del entorno gráfico, cuando el objeto de tipo *Marcador* está animado, y para este ejemplo —que es el caso utilizado en el código real de este proyecto— esto sucede cuando la adquisición está detenida.

4.4.1.4 Componente gráfico «ElementoSelector»

Es un control cuya función es idéntica a la de un botón de opción. Le diferencia de este en que contiene una imagen de fondo —que determina su tamaño—, un botón de opción y un control de texto que informa acerca de la opción que constituye el elemento. Se e agrupa con otros de su mismo tipo dentro de un control Selector, que solo muestra una pequeña cantidad de los elementos agrupados. Para visualizar el resto hay que deslizarlos sobre el área rectangular que representa el Selector. Su apariencia, en el proyecto es:

Ilustración 4.15: Apariencia del control ElementoSelector en este proyecto

Estos controles, junto con el Selector que les agrupa, se encuentran, únicamente, en el menú Adquisición. La parte privada de la clase que representa este control, y su constructor, son:

```
1  template<uint16_t CAPACIDAD> class Selector;//DECLARACIÓN ADELANTADA
2
3  class ElementoSelector : public Container
4  {
5  private:
6      Image imgFondo;
7      RadioButton botonSelector;
8      TextArea textoElemento;
9      template<uint16_t CAPACIDAD> friend class Selector;
10
11 public:
12     ElementoSelector():Container()
13     {
14         add(imgFondo);
15         add(botonSelector);
16         add(textoElemento);
17     }
18          . . .
19 };
```

Código 4.24: Atributos de la clase ElementoSelector

Necesita la declaración adelantada de la clase *Selector* que va a agrupar varios objetos *ElementoSelector*—línea 1—, ya que se trata de una clase amiga —línea 9— para que pueda acceder a las partes privadas de cada elemento que contenga. La clase *ElementoSelector* deriva desde *Container* para crear un nuevo control por la técnica de composición —ver sección 1.13.2—. Así, esta composición está constituida por una imagen de fondo del control —línea 6—, un botón de opción —línea 7—, que será mutuamente excluyente con los botones de opción de otros elementos de selector, agrupados bajo su mismo selector, y un control de texto —línea 8— que informará del significado de la opción. En la parte privada, se declara también, la clase *Selector* como amiga de la clase *ElementoSelector*, lo que permite al selector acceder a las partes privadas de los objetos de tipo *ElementoSelector*. Para que estos tres elementos estén visibles en el contenedor que los contiene, es necesario añadirlos a él. Esto se hace dentro de su constructor —líneas 14, 15 y 16—. Los métodos que dispone esta clase se limitan a configurar estos tres elementos. Así el método *ponFondo*, espera como argumento un identificador

de imagen —de la base de datos de imágenes— para que se cargue en el elemento *imgFondo*. El tamaño de esta imagen constituirá el tamaño del control. El método *ponBitmaps*, establece las cuatro imágenes que configuran el botón de opción interno —*botonSelector*—, posicionando estas imágenes es la zona izquierda del fondo, y el método *ponTexto*, para poner el texto referente a la opción. La declaración de objetos de este tipo se realiza como el resto de los controles, en la parte privada de la vista que los contiene:

```
1  class FftView : public View<FftPresenter>
2  {
3    public:
4         . . .
5    private:
6         . . .
7         ElementoSelector elementosFs[4];
8         . . .
9  };
```

Código 4.25: Declaración de objetos de tipo ElementoSelector

Mediante el array «*elementoFs*» se declaran cuatro objetos de tipo *ElementoSelector*.

```
1  void FftView::setupScreen()
2  {
3   . . .
4   for (int16_t i=0; i<4; i++)
5   {
6      elementosFs[i].ponFondo(BITMAP_FILASELECTOR_ID);
7      elementosFs[i].ponBitmaps(BITMAP_RADIO_BUTTON_UNSELECTED_ID,
8                                BITMAP_RADIO_BUTTON_UNSELECTED_PRESSED_ID,
9                                BITMAP_RADIO_BUTTON_SELECTED_ID,
10                               BITMAP_RADIO_BUTTON_SELECTED_PRESSED_ID);
11     elementosFs[i].ponColorTexto(Color::getColorFrom24BitRGB(0x34, 0x90, 0xCA));
12  }
13  elementosFs[0].setSelected(true);
14  elementosFs[0].ponTexto(TypedText(T_TEXTOFS1));
15  elementosFs[1].ponTexto(TypedText(T_TEXTOFS2));
16  elementosFs[2].ponTexto(TypedText(T_TEXTOFS3));
17  elementosFs[3].ponTexto(TypedText(T_TEXTOFS4));
18  . . .
19 }
```

Código 4.26: Configuración de los objetos de tipo ElementoSelector

Como se trata de un array, la configuración común de cada objeto se realiza dentro de un bucle *for* —líneas entre la 4 y la 12—, así, las imágenes de fondo y la de los botones de opción se pueden configurar de esta manera. Sin embargo, la configuración particular de cada elemento, se ha de realizar por separado. Este es el caso de los textos de cada opción —líneas de la 14 a la 17—. En la línea 13 se muestra el método *setSelected*, de la clase *ElementoSelector*, que pone al objeto que lo llama como marcado —seleccionado—. Como todos estos cuatro elementos de selector estarán contenidos en un mismo selector —no mostrado en el código, ver sección 4.4.1.5—, el llamar al método *setSelected,* produce la selección —marcado— del objeto que lo invoca y deja sin selección —desmarcado— el resto de objetos.

4.4.1.5 Componente gráfico «Selector»

Este control agrupa varios objetos de tipo *ElementoSelector* para producir la exclusión mutua entre ellos ya que cada uno se comporta como un botón de opción. Su apariencia es la de un marco en donde se visualizan alguno de los elementos que agrupa —para ver el resto se ha de producir el arrastre con el dedo sobre el control —. En este proyecto, su apariencia es:

Ilustración 4.16: Apariencia del control Selector en este proyecto

Este control, junto con los elementos que agrupa —*ElementoSelector*—, se encuentran, únicamente, en el menú Adquisición. Los atributos que constituyen esta clase son:

```
1  template<uint16_t CAPACIDAD>
2  class Selector : public Container
3  {
4  private:
5      uint16_t n_elementos;
6
7      Image imgFondo;
8      ScrollableContainer ContenedorDeslizable;
9      ListLayout ConenedorLista;
10     RadioButtonGroup<CAPACIDAD> grupoOpciones;
11
12 public:
13     . . .
14 };
```

Código 4.27: Atributos de la clase Selector

Se trata de una clase para crear un control por composición —hereda de la clase *Container*—. Se compone de una imagen de fondo —línea 7—, un contenedor deslizable —línea 8—, un contenedor tipo lista —línea 9—, donde se irán añadiendo los controles *ElementoSelector* en forma de lista, y que estará contenido en el contenedor deslizable para poder acceder a cualquier elemento de la lista aunque esta no quepa físicamente en el contenedor tipo lista —se accederá a los elementos mediante deslizamientos del control—. Finalmente, el control Selector, contiene un objeto de tipo *RadioButtonGroup* —grupoOpciones, línea 10—, que es el encargado de gestionar la exclusión mutua de los objetos *ElementoSelector* añadidos.

La declaración del objeto de tipo Selector se realiza en la parte privada de la vista que lo contiene:

```
1  class FftView : public View<FftPresenter>
2  {
3    public:
4        . . .
5    private:
6        . . .
7        Selector<4> contenedorElementosFs;
8        . . .
9  };
```

Código 4.28: Declaración de objeto de tipo Selector

Mediante un bucle de tipo *for*, el array «*elementoFs*» se declaran cuatro objetos de tipo *ElementoSelector*.

```
1  void FftView::setupScreen()
2  {
3   . . .
4   contenedorElementosFs.ponFondo(BITMAP_FONDOSELECTOR_ID);
5   contenedorElementosFs.setXY(5,2);
6   for (int16_t i=0; i<4; i++)
7   {
8      . . .
9      contenedorElementosFs.add(elementosFs[i]);
10  }
11  contenedorElementosFs.setRadioButtonSelectedHandler(BotonRadioSelecciondoCallback);
12  . . .
13 }
```

Código 4.29: Configuración del objeto de tipo Selector

Primero se le asigna la imagen de fondo —línea 4— y posición —línea 5—, luego se añada cada uno de los elementos de tipo *ElementoSelector* —array *elementosFs*, en línea 9—, y para finalizar, se le asigna un objeto *Callback* —*BotonRadioSelecciondoCallback*, línea 11— enlazado con el manipulador de usuario *Al_BotonRadioSelecciondo*, encargado de gestionar todos los eventos de botones de tipo opción —*RadioButtons*—:

```
1  void FftView:: Al_BotonRadioSelecciondo(const AbstractButton& radioButton)
2  {
3   . . .
4   if (&radioButton >= contenedorElementosFs.getRadioButton(0)/
5       && &radioButton <= contenedorElementosFs.getRadioButton(3))
6      {
7          uint8_t fs=contenedorElementosFs.getSelectedRadioButtonIndex();
8          presenter->obtenerModelo()->ponerMuestreo((DispositivoFFT::RelojMuestreador)fs);
9      }
10  . . .
11 }
```

Código 4.30: Manipulador de evento de usuario para el control Selector

Primeramente se comprueba que los botones de opción que se van a gestionar son los pertenecientes a los elementos de tipo *ElementoSelector* del objeto *contenedorElementosFs* —de tipo *Selector*—que se declaró previamente en la parte privada de la vista. Esta comprobación se hace en la línea 4. Luego, se recupera el índice del objeto de tipo *ElementoSelector* que disparó el evento —línea 7— y con este dato, se actualiza el modelo —línea 8—.

4.4.1.6 Componente gráfico «TBotonOcultado»

Este control se trata de una plantilla que espera como tipo indeterminado una clase botón —*Button*— o descendiente de esta, y cuyo cometido es mostrar y ocultar controles mediante la modificación paulatina de su transparencia —animación alfa—. Dispone de dos métodos principales, uno para iniciar la aparición del control —*iniciaAparicion*— y otro para iniciar la ocultación —*iniciaOcultacion*—. Para ello, gestiona una serie de temporizaciones como son: un retardo antes de iniciar la aparición, la duración de la aparición, un retardo antes de iniciar la ocultación y la duración de la ocultación. Todo ello es controlado por atributos internos. Posee dos objetos de tipo *GenericCallback* que posibilitan disparar eventos a la finalización de la animación (aparición y ocultación) y, de forma continuada, en cada actualización de pantalla —*tick*—, mientras está transcurriendo la animación. En este proyecto se ha utilizado para los botones de control de escala vertical y horizontal, y para ello

se ha usado como tipo indeterminado de la platilla el tipo *TPulsador*, que a su vez es otra plantilla —ver sección 4.4.1.1— y el tipo *Button* como tipo de esta última plantilla. Estos botones han de iniciar su aparición cuando se toca la zona de la gráfica —ver sección 4.4.1.2, control *TouchAreaMio*—, y su ocultación cuando la gráfica lleve un rato sin tocar, o se proceda a la realización de la cancelación de estos botones —se realiza la presión sobre ellos y la liberación fuera de sus límites—. Es, por tanto, este control que ocupa toda la zona gráfica y que es transparente —*TouchAreaMio*—, el encargado de iniciar la aparición y ocultación de estos botones. Este tipo de control puede agruparse para cancelar la ocultación de todos los controles del grupo cuando uno de ellos ha cancelado la ocultación —ver control *TAgrupacionBotonesOcultados*, en la sección 4.4.1.7—. Su apariencia es la siguiente:

Ilustración 4.17: Apariencia del control Selector en este proyecto

Para contener los atributos de este control, en vez de definir una parte privada —*private*–, se define una parte protegida —*protected*—, de esta forma, y ante una posible herencia por parte de una clase derivada, se pueda tener acceso a los atributos por parte de esta. Aunque por el momento esta clase es la hoja final de la jerarquía de herencia y por tanto, se podría haber utilizado la especificación privada. La declaración de atributos y demás elementos de la parte protegida son los siguientes:

```
template<class T>
class TBotonOcultado : public T
{
    public:
                . . .
    protected:
        template<class Tipo> friend class TAgrupacionBotonesOcultados;
        TAgrupacionBotonesOcultados<T>  *grupo;
        bool       ejecutandose;
        bool       ocultando;
        uint16_t   retardoInicioAparicion;
        uint16_t   duracionAparicion;
        uint16_t   retardoInicioOcultacion;
        uint16_t   duracionOcultacion;
        uint16_t   duracionEnUso;
        uint16_t   retardoEnUso;
        uint8_t    alfaInicial;
        uint8_t    alfaFinal;
        uint8_t    alfaFinalConfigurada;
        uint16_t   contador;
        EasingEquation Ecuacion;

        GenericCallback<const AbstractButton &, uint8_t>* AnimacionTerminadaCallback;
        GenericCallback<const AbstractButton &, uint8_t>* AnimacionEnCursoCallback;

        void AlAcabarPresentacion(void){. . .}
        void AlEjecutarPresentacion(void){. . .}
        virtual void handleClickEvent(const ClickEvent &event){. . .}
        virtual void handleTickEvent(){. . .}
};
typedef TBotonOcultado<Button>  BotonOcultado;
```

Código 4.31: Atributos de la clase TBotonOcultado

Los atributos que contienen los valores de las temporizaciones son: *retardoInicioAparicion* —temporización inicial, antes de la aparición—, *duracionApaicion* —tiempo que se utiliza en la aparición del control—, *retardoInicioOcultacion* —temporización inicial, antes de la ocultación—, *duracionOcultacion* —tiempo utilizado en la ocultación del control—. Estos tiempos pueden modificarse, para ello, la clase provee los métodos públicos de acceso a estos atributos. Para unificar el uso interno de estas temporizaciones la clase dispone de los atributos *retardoEnUso*, que contiene el retardo de aparición u el de ocultación —el que esté sucediendo en cada momento—, y *duracionEnUso*, que contiene tanto la duración inicial antes de la aparición o antes de la ocultación. Para configurar los niveles de transparencia están los atributos *alfaInicial*, *alfaFinal* y *alfaFinalConfigurada*. El primero contiene el valor de transparencia desde el que se parte, que será igual al valor del atributo *Alpha* del control del tipo indeterminado de la plantilla que tiene en cada proceso —de aparición o de ocultación—. El segundo contiene el valor final de transparencia de la animación, que será igual a cero, para el caso de la ocultación o igual al valor del atributo *alfaFinalConfigurada* —tercer atributo de transparencia— para el caso de la aparición. El tercer atributo se ha de configurar, mediante el su método de acceso, para contener el valor de transparencia que se quiere cuando el control aparezca. Existen otros atributos para el control del funcionamiento interno de la clase como es *ejecutandose*, que está a valor verdadero mientras está la animación en marcha, *ocultando*, que está a verdadero si la animación es de ocultación, *contador*, que contiene el valor de cada paso de animación y *Ecuacion*, que contiene la ecuación utilizada en la animación —ver sección 1.15—. Luego están los objetos de tipo *GenericCalback*, como es *AnimacionTerminadaCallback*, pensado para producir el disparo de evento cuando la animación ha terminado, o *AnimacionEnCursoCallback*, que gestiona el disparo continuado de evento mientras la animación está en curso. Ambos tiene dos argumentos, el primero, la dirección del objeto desde donde se dispara el evento, para poder identificarlo en el manipulador de eventos de usuario y de esa forma, ejecutar el bloque de código correcto, y el segundo, un valor entero sin signo de 8 bits, puede ser 1, si se está produciendo la aparición, o 2, si es la ocultación.

La clase posee una serie de métodos en la parte protegida —sin acceso desde fuera de la clase—, ya que son para manejar el funcionamiento interno. Así, *handleClickEvent*, que es llamado por la capa HAL cuando el control es tocado, gestiona la parada de la ocultación del control tocado así como del resto de controles que pertenezcan

a su grupo, reanudando la ocultación cuando es liberado. *handleTickEvent* es el método ejecutado por la capa HAL en cada actualización de pantalla cuando el control ha sido registrado para ello. Es en este método es donde se generan las animaciones. Para ello, los métodos que inician las animaciones, tienen como una de sus funciones registrar al componente para recibir eventos de temporización —que la capa HAL ejecute *handleTickEvent*—. Dentro de *handleTickEvent* se hace uso de los métodos *AlEjecutarPresentacion* y *AlAcabarPresentacion*. El primero de ellos, tiene como misión ejecutar el objeto *AnimacionEnCursoCallback* para llamar al manipulador de evento de usuario mientras se está ejecutando la animación. El segundo, ejecuta el objeto *AnimacionTerminadaCallback*, que ejecuta el manipulador de usuario cuando ha terminado la animación además si el control acaba de aparecer, lo pone como tocable —puede recibir pulsaciones desde la pantalla táctil—, o si el control acaba de ocultarse, lo pone como no visible y no tocable. El método *handleTickEvent*, al terminar la animación, quita del registro de eventos de temporización el propio control y son los métodos de iniciación los que vuelven a registral el control. El método *iniciaAparicion*, además, pone el control como visible —aunque inicialmente el control sea totalmente transparente—.

La gestión de la cancelación de la ocultación se realiza en el método *handleTickEvent* de la siguiente forma:

```
1  virtual void handleClickEvent(const ClickEvent &event)
2    {
3      T::handleClickEvent(event);
4      if (ocultando)
5       {
6         if (event.getType()==ClickEvent::PRESSED)
7          {
8            contador=0;
9            Application::getInstance()->unregisterTimerWidget(this);
10         }
11        else
12           Application::getInstance()->registerTimerWidget(this);
13        if (grupo)grupo->evitaOcultacionGrupo(*this, event);
14      }
15   }
```

Código 4.32: Método handleClickEvent de la clase TBotonOcultado

Lo primero que hace este método-evento, es llamar a su homónimo en la clase de la que desciende —en este caso T, el tipo indeterminado, en línea 3— para que se gestione el normal funcionamiento de este evento. Luego, solo ejecuta código, para esta clase, si se está produciendo la ocultación —línea 4—. Si es así, detecta si se ha presionado sobre este botón —línea 6—, en tal caso, pone el contador de animación a cero —línea 8—, para que la siguiente animación se inicie desde el principio, y quita al componente del registro de recepción de eventos de temporización que evita que continúe la animación de ocultación —línea 9—. El objeto de tipo *TouchAreaMio* —*areaTocable*, ver sección 4.4.1.2—, donde residen estos botones, es quien inicia de nuevo su aparición, ya que se ha tocado sobre esta área —*areaTocable*—, y es ella la responsable del inicio de la aparición —al ser tocada el área— y la ocultación —al liberar la presión—. Si se produce la liberación sobre el botón, se vuelve a registrar el control para recibir eventos de temporización. En cualquiera de los casos, si el control pertenece a un grupo, se envía a este grupo el control que ha producido el evento y la información sobre el evento —línea 13—, para que el objeto que representa el grupo, cancele la ocultación del resto de objetos de tipo *TBotonOcultado* que pertenecen a ese grupo. El objeto que representa el grupo, básicamente, hace lo mismo que este evento pero para el resto de componentes del grupo —ver sección 4.4.1.7—.

La declaración, en la parte privada de la vista, junto con el objeto de la clase que agrupa a varios objetos de tipo *TBotonOcultado*, es la siguiente:

```
1  class FftView : public View<FftPresenter>
2  {
3  public:
4      . . .
5  private:
6      . . .
7      TBotonOcultado< TPulsador< Button > >botonOcultado_masV;
8      TBotonOcultado< TPulsador< Button > >botonOcultado_menosV;
9      TAgrupacionBotonesOcultados<TPulsador< Button > > grupoOcultadoV;
10     . . .
11 };
```

Código 4.33: Declaración de la plantilla de tipo TBotonOcultado

En la línea 7 se declara el objeto *botonOcultado_masV*, de tipo *TBotonOcultado*, especificando como tipo de plantilla *TPulsador*, otra plantilla que tiene como tipo *Button*. La clase Button es modificada por TPulsador para comportarse como un pulsador —que detecte presiones y no solo presiones con la consiguiente liberación—, y esta es modificada por *TBotonOcultado* para que además se pueda ocultar o mostrar de forma progresiva. De igual modo se declara el objeto *botonOcultado_menosV* —línea 8—. Para poder agrupar estos dos controles de tal forma que cuando se cancele la ocultación en uno de ellos se cancele la del otro y viceversa, se crea el objeto *grupoOcultadoV* —línea 9— de la clase *TAgrupacionBotonesOcultados* —ver siguiente sección 4.4.1.7—, que tiene que tener el mismo tipo en la plantilla que el que tienen los botones de ocultación.

La configuración de estos objetos se realiza dentro del método *setupScreen* de la vista de la siguiente forma:

```
1  void FftView::setupScreen()
2  {
3      . . .
4      botonOcultado_masV.setBitmaps(Bitmap(BITMAP_MAS_NOPRES_ID),Bitmap(BITMAP_MAS_PRES_ID));
5      botonOcultado_masV.setXY(0,0);
6      botonOcultado_masV.ponerDuracionAparicion(1);
7      botonOcultado_masV.ponerDuracionOcultacion(30);
8      botonOcultado_masV.ponerRetardoInicioOcultacion(45);
9      botonOcultado_masV.setVisible(false);
10     . . .
11     grupoOcultadoV.agrega(&botonOcultado_menosV);
12     grupoOcultadoV.agrega(&botonOcultado_masV);
13     grupoOcultadoV.setAlpha(220);
14     . . .
15 }
```

Código 4.34: Configuración del objeto de tipo TBotonOcultado

Inicialmente se configura el aspecto del objeto *botonOcultado_masV*, determinando las imágenes que tendrá el control al ser presionado y liberado —línea 4— y su posicionamiento —línea 5—. Luego se configuran las temporizaciones de las animaciones como son: la duración de la animación de la aparición —línea 6—, la duración de la animación de la ocultación —línea 7— y el retardo inicial a la ocultación —línea 8—. El no especificar el retardo inicial a la aparición significa que se usa el valor por defecto de la clase que es de cero *ticks*. Como, inicialmente se quiere que el control esté ocultado, se pone su propiedad *visible* a valor falso —línea 9—. El objeto *botonOcultado_menosV* —no mostrado en el código— se configura de forma similar a como se ha hecho con *botonOcultado_masV*. El objeto que agrupa a los dos objetos anteriores —*grupoOcultadoV*—, para que actúen de forma solidaria a la hora la cancelación de la ocultación, agrega cada objeto al grupo mediante su método *agrega* —líneas 11 y 12— pasándole como argumento la dirección del objeto a añadir. El objeto de agrupamiento puede, entre otras cosas, cambiar la transparencia de todos los objetos del grupo —línea 13—.

4.4.1.7 Componente gráfico « TAgrupacionBotonesOcultados»

Al igual que el control *TBotonOcultado*, se trata de una plantilla. Su cometido es la de agrupar varios objetos de tipo *TBotonOcultado* para gestionar su funcionamiento de de forma conjunta. Así, dispone de los métodos *iniciaAparicion* e *iniciaOcultacion*, que inician la aparición u ocultación, respectivamente, de todos los objetos que conforman el grupo. También gestiona la cancelación de la ocultación de componentes del grupo. El tipo indeterminado de la plantilla, cuando se crea el objeto, ha de ser del mismo tipo que el de la plantilla de los objetos de tipo *TBotonOcultado* a los que pretende agrupar. Además, posee el par de métodos *setAlpha* y *getAlpha*, que cambian o recuperan, respectivamente, el valor conjunto de transparencia de los componentes del grupo. Es un componente sin representación gráfica y está definido en un solo archivo de cabecera: *TAgrupacionBotonesOcultados.hpp*. Sus atributos son los siguientes:

```
1  template<class T>
2   class TBotonOcultado;//declaración adelantada
3
4   template<class Tipo>
5   class TAgrupacionBotonesOcultados
6   {
7       public:
8          TAgrupacionBotonesOcultados():
9                    n_elementos(0), alfa(255)
10          {}
11          . . .
12      private:
13          uint8_t n_elementos;
14          TBotonOcultado<Tipo> *agrupados[MAX_OCULTADOS];
15          uint8_t alfa;
16  };
```

Código 4.35: Atributos de la clase TAgrupacionBotonesOcultados

Ya que va a agrupar objetos de tipo *TBotonOcultado*, necesita un array de punteros a estos objetos como atributo —*agrupados*, línea 14—, y otro atributo que albergue el número de elementos del grupo —*n_elementos*, línea 13—. También dispone del atributo alfa —línea 15— destinado a contener el nivel de transparencia de todos los elementos del grupo. Estos dos últimos atributos son inicializados en el constructor de la clase —líneas de la 8 a la 10— a los valores respectivos de cero —inicialmente sin elementos agrupados— y 255 —opacidad total—. El tipo de los elementos del grupo —tipo plantilla—, se declara de forma adelantada a la definición de la clase *TAgrupacionBotonesOcultados* —líneas 1 y 2—, ya que es un tipo que internamente maneja esta clase —línea 14—.

El código del método que se dedica a añadir elementos al grupo —*agrega*— es el siguiente:

```
1  void agrega(TBotonOcultado<Tipo> *elemento)
2   {
3    if (n_elementos<=MAX_OCULTADOS)
4     {
5      agrupados[n_elementos]=elemento;
6      elemento->ponAgrupamiento(this);
7      n_elementos++;
8     }
9   }
```

Código 4.36: Método «agrega» de la plantilla de tipo TAgrupacionBotonesOcultados

Su argumento es el puntero del objeto a añadir —línea 1—. Solo se procede a añadir un nuevo elemento al grupo si no se rebasa la capacidad del array *agrupados* —constante MAX_OCULTADOS definida a 8 en el archivo de cabecera de la clase *TAgrupacionBotonesOcultados*—. Si no es así, el nuevo elemento es añadido al array —

línea 5—, se envía al elemento recién añadido, la dirección del objeto que representa el grupo al que pertenece —línea 6—, y se incrementa el atributo que indica el número de elementos agrupados —línea 7—.

Para gestionar la evitación de la cancelación de la ocultación de todo el grupo, posee el método *evitaOcultacionGrupo*, que es llamado desde el evento *handleClickEvent* de objeto perteneciente al grupo, que inicia la cancelación de la ocultación —ver sección 4.4.1.6—. Su código es el siguiente:

```
1  void evitaOcultacionGrupo(TBotonOcultado<Tipo> &componente, const ClickEvent &evento)
2  {
3    for (int i=0; i<n_elementos; i++)
4     {
5      if (agrupados[i]!=&componente)
6      {
7       if (evento.getType()==ClickEvent::PRESSED)
8       {
9         agrupados[i]->contador=0;
10        Application::getInstance()->unregisterTimerWidget(agrupados[i]);
11      }
12      else
13        Application::getInstance()->registerTimerWidget(agrupados[i]);
14     }
15    }
```

Código 4.37: Método «evitaOcultacionGrupo» de la plantilla de tipo TAgrupacionBotonesOcultados

Realiza exactamente la misma función que el método *handleClickEvent* del elemento del grupo que lo invoca, pero para el resto de elementos del grupo —ver sección 4.4.1.6—

Para ver cómo se declara un objeto de este tipo en la vista, así como se configura dentro del método *setupScreen* de la vista, ver la explicación de la plantilla *TBotonOcultado* en la sección 4.4.1.6, es aconsejable ver también la sección 4.4.1.2 dedicada al control *TouchAreaMio.*

4.4.1.8 Componente gráfico «BotonEstados»

La pretensión de este componente es el acceso rápido a funcionalidad relacionada. De hecho, en algunos casos, un solo botón de este tipo puede sustituir todo un menú. Para ello, este botón tiene una serie de estados programados asociados a cierta acción. Cada vez que se produce una pulsación se va cambiando de estado de forma cíclica. Cada estado, dentro del botón, aparece como un nombre —una cadena que identifica el estado— que al entrar en él puede dispararse cierta acción. Además de las cadenas de texto de los estados, posee un nombre identificativo del botón. Su aspecto es el siguiente:

Ilustración 4.18: Apariencia del control BotonEstados en este proyecto

Por ejemplo, en la figura se muestra, entre otros, el botón de estado llamado «ESQUEMAS». Mediante su pulsación de forma repetida, se va cambiando a los estados «Jet», «Hielo», «Fuego», «Oceano», «Arco iris», «Invierno» y «Verano». Este botón tiene la misma funcionalidad que el menú «ESQUEMAS», con el inconveniente

que para llegar a un esquema concreto, se ha debido pasar antes por el resto de esquemas. Es por ello que este tipo de botones se engloban dentro del menú «RAPIDO». Es un componente que solo dispone de archivo de cabecera, llamado BotonEstados.hpp. La parte privada y protegida de la clase es la siguiente:

```
1  class BotonEstados : public Container
2  {
3   protected:
4
5     Button boton;
6     TextArea texto;
7     TextAreaWithOneWildcard nombre_estado_actual;
8     Unicode::UnicodeChar cadena[15];
9     uint8_t estado;
10    uint8_t n_estados;
11    char **nombresEstados;
12
13  private:
14
15    void pasaAlSiguienteEstado(void)
16     {
17       estado+=1;
18       if(estado>=n_estados)
19          estado=0;
20       Unicode::strncpy(cadena, nombresEstados[estado], 15);
21       nombre_estado_actual.resizeToCurrentText();
22
23       nombre_estado_actual.setPosition(getWidth()/2-nombre_estado_actual.getTextWidth()/2,
24       getHeight()/4-nombre_estado_actual.getTextHeight()/2,
25       nombre_estado_actual.getTextWidth(),boton.getHeight());
26
27       nombre_estado_actual.invalidate();
28     }
29
30    Callback<BotonEstados, const AbstractButton&> areaTocableClickedCallback;
31    GenericCallback< const BotonEstados & > * vistaCallback;
32 };
```

Código 4.38: Definición de la clase BotonEstados

Es un componente nuevo mediante composición basado en un contenedor —línea 1—. Como elementos constitutivos de la composición está un botón —línea 5—, que ocupará toda el área del componente, un texto —línea 6—, que identificará al botón, y un texto variable —línea 7— que mostrará los textos correspondientes a los estados en cada pulsación. Como otros atributos constituyentes del componente, están el *estado* —línea 9—, que identificará numéricamente el estado en que se encuentra; el número de estados —*n_estados* en línea 10—, que establece la cantidad de estados que tendrá el componente; y el puntero a cadenas —nombreEstados—, que apuntará al array que contendrá los textos para cada estado. Todo ello ha sido englobado en la parte protegida de la clase con la futura intención de ser accedida por parte de clases derivadas —aunque no se ha realizado en este proyecto—. Por otro lado está la parte privada, donde está el código de solo acceso por parte de la clase. En ella se localiza el método *pasaAlSiguienteEstado*, que será llamado en cada pulsación del botón. Este método asegura el tránsito cíclico entre estados pasando al primer estado una vez llegado al último, así como mostrar la cadena correspondiente al estado. También posee dos atributos, el objeto *areaTocableCallback*, de tipo *Callback* y el puntero a objeto de tipo *GenericCallback* llamado *vistaCallback*. El primero de ellos tiene por objeto redirigir el evento *handleClickEvent*, del botón interno del componente, a una método que actué de manipulador de usuario pero de forma interna del componente. El segundo apuntará al objeto *Callback* que el usuario determine para ejecutar su manipulador de evento. Este puntero es desreferenciado en el manipulador interno del componente, con lo que la finalidad de estos dos atributos es transmitir el evento *Click* del botón interno del componente al exterior —para que pueda disparar un manipulador de usuario en la vista—. El código relacionado con este mecanismo es el siguiente:

```
1  class BotonEstados : public Container
2  {
3      . . .
4      public:
5          BotonEstados():
6                  Container(),
7                  estado(0),
8                  n_estados(2),
9                  areaTocableClickedCallback(this, &BotonEstados::areaTocableClicked),
10                 vistaCallback(0)
11         {
12             add(boton);
13             add(texto);
14             add(nombre_estado_actual);
15         }
16
17          void setCallBack(GenericCallback< const BotonEstados &  >& callback)
18         {
19             vistaCallback = &callback;
20         }
21
22
23         void areaTocableClicked(const AbstractButton& source)
24         {
25             pasaAlSiguienteEstado();
26             if (vistaCallback && vistaCallback->isValid())
27             {
28                 vistaCallback->execute(*this);
29             }
30         }
31     . . .
32 };
```

Código 4.39: Mecanismo de evento del la clase BotonEstados

En el constructor se inicializa el atributo *areaTocableClickedCallback* para que ejecute el método interno *areaTocableClicked* —línea 9—. Este método —línea 23—, primeramente llama al método *pasaAlsiguienteEstado*, ya que se ha producido la pulsación del botón interno y se necesita pasar al siguiente estado. Luego, desreferencia el puntero *vistaCallback* —línea 28— para que ejecute el objeto de tipo *Callback* de la vista que representa el manipulador de evento de usuario, y de esta forma se ha podido «retransmitir» el evento del botón interno hacia el exterior de la clase que lo contiene —*BotonEstados*—. Esta desreferencia del puntero *vistaCallback* solo se produce si apunta a algún objeto de tipo *Callback* y este último tiene sus punteros internos al objeto que contiene el manipulador de usuario y al propio método, respectivamente, válidos —línea 26—. Para cargar el puntero *vistaCallback*, la clase *BotonEstados* posee el método *setCallBack* —línea 17—.

La clase *BotonEstados* posee otros métodos públicos necesarios para poder configurarlo. Estos métodos son: *inicializa*, *ponNobresEstados*, *ponTexto*, *ponN_estados*, *ponEstado* y *obtenEstado*. El método *inicializa* permite establecer lo bitmaps para cada estado gráfico del botón —presionado y liberado—, así como especificar el número de estados que tendrá el botón —este argumento es opcional—. Para especificar los bitmaps solo se necesita sus identificadores de la base de datos de bitmaps. El método *ponNombresEstados* establece los nombres que tendrá cada estado. Para ello hay que pasarle como argumento, el array con los nombres de cada estado —el índice del array corresponde con el número que tiene cada estado—. El método *ponTexto*, establece el texto descriptivo que tendrá el botón. Para ello es necesario pasarle como argumento el identificador de cadena de la base de datos de textos de *TouchGFX*. El método *ponN_estados* permite establecer el número de estados pasado en su argumento. Si se ha especificado el número de estado como tercer argumento del método *inicializa*, no es necesario llamar al método *ponN_estados.* El método *ponEstado* permite establecer cuál es el estado del botón, sin necesidad de recorrer cíclicamente los estados pulsando repetidamente el botón. Finalmente está el método *obtenEstado* que retorna el valor numérico del estado en se encuentra el botón.

En la siguiente porción de código de la cabecera de la vista, *FftView.hpp* —que corresponde al código real de este proyecto—, se muestra el caso de la declaración de objetos de tipo *BotonEstados*:

```
1  class FftView : public View<FftPresenter>
2  {
3   public:
4      FftView():
5                  . . .
6                 BotonEstadosCallback(this, &FftView::Al_BotonEstados),
7                 . . .
8                 { . . .}
9           . . .
10     void Al_BotonEstados(const BotonEstados &boton);
11     . . .
12  private:
13     . . .
14     #define N_BOTONES_ESTADO 7
15     BotonEstados BotonesEstado[N_BOTONES_ESTADO];
16     . . .
17     Callback<FftView, const BotonEstados &> BotonEstadosCallback;
18 };
```

Código 4.40: Declaración del objeto de tipo BotonEstados

En la línea 15 se declara un array de controles de tipo *BotonEstados* con un tamaño de siete elementos, definido por la constante N_BOTONES_ESTADO —línea 14—. Para la manipulación de las pulsaciones de estos botones, y en consecuencia, la gestiones de estados de los mismos, se establece el manipulador de usuario *Al_BotonEstados* —línea 10—, cuyo argumento es el componente que genera el evento. Para poder enlazar este método con el evento de cada botón, se crea el objeto *BontonEstadosCallback* —línea 17—, donde se concreta el tipo indeterminado de la clase de la vista que va a contener el manipulador y un primer argumento que identifica el componente que genera el evento. En la lista de inicialización del constructor de la vista se construye el objeto *BotonEstadosCallback* —línea 6—, donde se especifica la dirección del objeto de tipo FftView y la dirección de su método que actuará de manipulador de evento. Con ello se ha enlazado el método que actuará como manipulador de evento con el objeto de tipo *Callback*. Se necesita realizar el enlace entre el objeto de tipo *BotonEstados* con este objeto *Callback* para que el proceso de manipulación de eventos. Esto, como en casos anteriores, se realiza en el archivo fuente de la vista —*FftView.cpp*—, en concreto en el método *setupScreen* de la misma, donde además, se produce el resto de configuración del componente:

```
1  void FftView::setupScreen()
2  {
3    . . .
4    for(uint8_t i=0; i<N_BOTONES_ESTADO; i++)
5     {
6      BotonesEstado[i].iniciializa(BITMAP_BOT_EST_NOPRES_ID,BITMAP_BOT_EST_PRES_ID,N_GAMAS);
7      BotonesEstado[i].setCallBack(BotonEstadosCallback);
8      . . .
9     }
10   . . .
11   static char *vent[5]={(char *)"RECTAN.",(char *)"HANNING",
12                         (char *)"HAMMING",(char *)"B.HARRIS",(char *)"BARTLETT"};
13   BotonesEstado[0].ponNombresEstados(vent);
14   BotonesEstado[0].ponTexto(TypedText(T_VENTANAS));
15   BotonesEstado[0].ponN_estados(5);
16   BotonesEstado[0].ponEstado(3);
17   . . .
18 }
```

Código 4.41: Configuración del objeto de tipo BotonEstados

El ejemplo mostrado corresponde al botón de estado que gestiona el cambio de ventana de la FFT. La configuración de los objetos *BotonEstados*, contenidos en el *array BotonesEstado*, se realiza, por un lado, de forma conjunta —líneas de la 4 a la 9— y por otro, de forma particular —líneas de la 11 a la 18—. En la parte conjunta se aprovecha el hecho de estar los objetos contenidos en el array, configurando los *bitmps* que tendrán

todos los botones —línea 6— y el manipulador que todos compartirán —línea 7—. Es en la determinación del manipulador donde se estable el enlace que faltaba entre el objeto de tipo *Callback* y los objetos de tipo *BotonEstados*. En la parte particular, para el objeto de tipo *BotonEstados*, contenido en el índice cero del array —el mostrado en la porción del código anterior—, se establecen las cadenas que nombrarán cada estado —líneas 11 y 12—, y se añaden al objeto —línea 13—. Luego se especifica el identificador de cadena, dentro de la base de datos de cadenas de TouchGFX, que se mostrará, dentro del componente para identificarlo y diferenciarlo del resto. Finalmente se especifica el número de estados —que ha de coincidir con el número de cadenas pasadas— y qué estado será el inicialmente mostrado. Este estado debe coincidir con el almacenado en el modelo para que todo funcione correctamente —ver sección 4.6.2—.

El manipulador de evento debe identificar, inicialmente, el objeto que disparó el evento para luego tratar que se hace en cada uno de sus estados:

```
1  void FftView::Al_BotonEstados(const BotonEstados &boton)
2  {
3   if(&boton == &BotonesEstado[0])
4    {
5     switch(presenter->obtenerModelo()->obtenerVentana())
6      {
7       case DispositivoFFT::RECTANGULAR:
8            presenter->obtenerModelo()->ponerVentana(DispositivoFFT::HANNING);
9            break;
10      case DispositivoFFT::HANNING:
11           presenter->obtenerModelo()->ponerVentana(DispositivoFFT::HAMMING);
12           break;
13      case DispositivoFFT::HAMMING:
14           presenter->obtenerModelo()->ponerVentana(DispositivoFFT::BLACKMAN_HARRIS);
15           break;
16      case DispositivoFFT::BLACKMAN_HARRIS:
17           presenter->obtenerModelo()->ponerVentana(DispositivoFFT::BARLETT);
18           break;
19      case DispositivoFFT::BARLETT:
20           presenter->obtenerModelo()->ponerVentana(DispositivoFFT::RECTANGULAR);
21           break;
22     }
23    }
24   . . .
25 }
```

Código 4.42: Manipulador de evento del objeto de tipo BotonEstados

En este caso concreto, referente al botón de estado que se encarga de seleccionar la ventana utilizada en la FFT, se pregunta al modelo sobre la ventana actualmente en uso —línea 5—. Dependiendo de la ventana que se esté utilizando, se pasa a la siguiente, en el orden que coincide con la secuencia de cadenas pasadas al objeto para servir de nombres de estado.

4.4.1.9 Componente gráfico «ElementoMenu»

Este componente trata de ser lo que su nombre indica: ser un elemento que compone un menú. Sin embargo, es posible su utilización de forma individual. En este sentido puede ser visto como un control que despliega un panel que puede contener otros componentes. Por ello, se trata de un nuevo componente creado a partir de un contenedor —sección 1.13.2— y su estructura es similar a la mostrada en la siguiente figura:

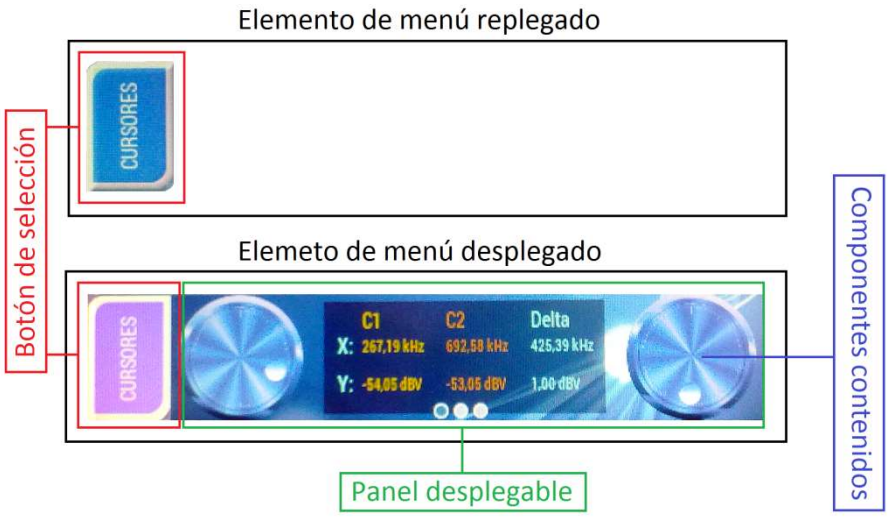


Ilustración 4.19: Apariencia del objeto de tipo ElementoMenu

Contiene un componente de tipo botón de enclavamiento —*ToggleButton*—, como parte del componente total que recibe la selección por parte del usuario; un componente de texto —*TextArea*–, que muestra un texto sobre el botón anterior; un componente de tipo contenedor —*Container*— que actúa como panel desplegable y un componente de tipo imagen —*Image*— de sirve de fondo del panel anterior.

Posee un comportamiento por defecto —cuando trabaja de forma individual— que consiste el desplegar el panel —mediante una animación— al ser presionada la parte botón de este componente. Este panel está inicialmente oculto, y al tratarse de un contendor, puede albergar otros componentes internamente. Cuando se vuelve a pulsar sobre la zona del botón, el panel se repliega para finalmente ocultarse. Este modo de actuar permite acceder a diversos controles sin que estos estén ocupando espacio en pantalla cuando no son utilizados. El propio botón, al ser de tipo enclavamiento, permite indicar cuando el panel está desplegado —botón enclavado—, y cuando está ocultado —botón desenclavado—. A pesar de ello, está diseñado para tener un comportamiento algo diferente cuando es agrupado junto a otros elementos de menú, formando un menú. Este agrupamiento se realiza a través de otro componente llamado menú —ver siguiente sección—. Para ello posee propiedades que se utilizan exclusivamente para este acontecimiento.

La parte protegida de la declaración de la clase es la siguiente:

```
1  namespace touchgfx
2  {
3      class Menu;//declaración adelantada
4
5      class ElementoMenu : public Container
6      {
7          public:
8              ElementoMenu();
9          . . .
10         protected:
11             Image imagenFondo;
12             ToggleButton botonMenu;
13             Container contenido;
14             TextArea texto;
15
16             uint16_t longitudDesplegado;
17             Orientacion orientacion;
18
19             bool ejecutandose;
20             uint16_t   contador;
21             uint16_t   retardo;
22             uint16_t   duracion;
23             EasingEquation Ecuacion;
24
25             int8_t id;
26             bool autodespliegue;
27
28             friend class Menu;
29             Menu *menu;
30
31             Callback<ElementoMenu, const AbstractButton&> botonMenuCallback;
32             GenericCallback< const ElementoMenu&, bool >* MenuSeleccionadoCallback;
33             GenericCallback< const ElementoMenu&, bool >* AnimacionMenuTerminadaCallback;
34             virtual void handleTickEvent();
35
36     };
37 }
```

Código 4.43: Declaración de la clase ElementoMenu

Esta declaración se realiza dentro del archivo de cabecera *ElementoMenu.hpp*. Está englobada dentro del espacio de nombres touchgfx para poder usar clases y variables, correspondiente a este espacio de nombres, sin que se deba especificar el mismo seguido del operador de resolución de alcance cada vez que se accede a uno de ellos. En la parte protegida dispone de un primer grupo de atributos que caracterizan el componente en sí y que ya han sido mencionados. Estos son: *imagenFondo* —en línea 11, se trata de la imagen de fondo del panel desplegable—, *botonMenu* —en la línea 12, es el único elemento visible cuando el panel no está desplegado y en el que se actúa para desplegar/replegar el panel—, *contenido* —en la línea 13, panel que se despliega y repliega al ser presionado el botón, y que contiene otros componentes—, y *texto* —en línea 14, se trata del texto que aparece dentro del botón—. El segundo grupo de atributos —líneas 16 y 17—, corresponden a aquellos que modifican el tipo de visualización del componente. Así *longitudDesplegable*, hace referencia a lo largo que es la zona del panel —que si no se especifica explícitamente, su valor es el de la longitud de la imagen utilizada de fondo—. La altura del panel desplegable siempre va a ser la del botón del componente global. Por otro lado, el atributo *orientación*, determina si el elemento de menú se va a mostrar horizontal o verticalmente. Esto implica que el panel se despliegue acorde con el valor de este atributo. También influye en la orientación del texto mostrado en el botón. El tipo de este atributo es *Orientacion*, un tipo enumerado que contiene los valores de *HORIZONTAL* Y *VERTICAL*. El tercer grupo de atributos está dedicado a controlar la animación del componente cuando es seleccionado o deseleccionado. Así duración —línea 22— estable la duración total de la animación en *ticks* del entorno gráfico. El atributo que se encarga de contar los *ticks* que han trascurrido en la animación es *contador* —línea 20—. El retardo inicial, antes de comenzar la animación, está regulado por el atributo *retardo* —

línea 21—. El atributo *ejecutándose* —línea 19— adquiere el valor de verdadero cuando la animación está en curso y falso en caso contrario. Es útil para impedir que la zona del botón sea insensible a pulsaciones mientras la animación se está ejecutándose. Finalmente, para este grupo, está el atributo *Ecuacion* —línea 23— de tipo *EasingEquation* —ver sección 1.15— que gobierna el movimiento durante la animación. En el cuarto grupo están los atributos que caracterizan el comportamiento en grupo. Así *autodespliegue* —línea 26—, atributo booleano, que es verdadero por defecto, indica si se produce el despliegue del panel nada más dar al botón o no. De forma individual, como este atributo es verdadero, se produce el despliegue del panel al presionar el botón; pero cuando este componente es parte de un menú, esta variable pasa a ser falsa para que el componente que controla todos los elementos del menú —clase tipo *Menu*, ver siguiente sección— pueda realizar una animación conjunta de todos los elementos de menú, y cuando haya acabado esta animación llamar al método de la clase *ElementoMenu* que inicia el despliegue de su panel —el método que inicia este despliegue se llama *iniciaAnimacion*—. Por otro lado está el atributo *id* —línea 25—, que sirve para identificar al elemento de menú dentro del menú, es decir cuando pertenece a un grupo de elementos de menú. Este identificador coincidirá con el índice de un *array* de punteros a la clase *ElementoMenu* que dispone la clase *Menu*. El quinto grupo de atributos son los relacionados con el grupo al que pertenece el elemento de menú. Así el atributo *menu* —línea 29—, se trata de un puntero al objeto de tipo *Menu* que gestiona este elemento de menú de forma conjunta con el resto de elementos de menús que constituyen el mismo grupo. Por otro lado está la declaración de la clase *Menu* como amiga de esta —línea 28— lo que posibilita que un objeto de la clase *Menu* pueda acceder a las partes privadas y protegidas de un objeto de la clase *ElementoMenu*. Esto posibilita que el objeto *Menu*, al que pertenece un determinado objeto *ElementoMenu*, pueda cambiar el valor del atributo *autodespligue* a falso, y realizar el despliegue del panel del objeto *ElementoMenu* cuando sea preciso. Finalmente, está el grupo de atributos relacionados con eventos. Así se redefine el atributo *handleTickEvent* —línea 34— para que este componente pueda gestionar los eventos de temporización para crear la animación. Para gestionar internamente el evento *Click* del botón que compone el componente, se crea el objeto de retrollamada *botonMenuCallback* —línea 31—. Este objeto llamará al método *AlBotonMenu* que, entre otras cosas, transmite el evento al exterior del objeto a través del puntero de retrollamada *MenuSeleccionadoCallback* —línea 32—, lo que posibilita que el usuario de respuesta mediante un manipulador de evento. El puntero *AnimacionMenuTerminadaCallback* —línea 33— es desreferenciado para llamar al manipulador de evento de usuario cuando la animación ha concluido.

Para preservar el carácter de encapsulamiento de la clase, todos los atributos están protegidos, lo que implica que no pueden ser directamente accedidos desde fuera de la misma, y solo se tiene acceso a ellos de forma interna o desde clases heredadas. Para poder obtener y modificar el valor de estos atributos, se suministran una serie de métodos del tipo *obtenXXX* y *ponXXX* para leer y modificar el atributo, respectivamente —las XXX hacen referencia al nombre del atributo en cuestión—. Sin embargo no todos los atributos son accedidos de este modo. Así los atributos de las clases que constituyen el componente —botón, imagen de fondo, texto en botón y panel desplegable— se modifican mediante los métodos *inicializa* —donde se especifican los dos *bitmaps* correspondientes al estado del botón—, *ponTexto* —donde se especifica el objeto de tipo texto que aparecerá encima del botón—, o *ponImagenFondo* —donde se especifica el identificador del *bitmap* de la base de *bitmaps* que se usará como fondo del panel desplegable. Como la clase *ElementoMenu* deriva de la clase *Container* —línea 5— se redefinen los métodos *add*, *remove* y *removeAll* —que pertenecen a la clase Container— para que los elemento añadido y borrados no se añadan directamente al componente, sino que lo hagan al panel desplegable —objeto interno *contiene*—. Los punteros *MenuSeleccionadoCallback* y *AnimacionMenuTerminadaCallback*, son cargados con los objetos de retrollamada de usuario mediante los métodos respectivos de *setSeleccionado* y *setTerminado*. Para finalizar con los métodos definidos en el archivo de cabecera, se muestra a continuación el código de uno ya nombrado, *iniciaAnimacion*:

```
1  class ElementoMenu : public Container
2   {
3     public:
4       ElementoMenu();
5       . . .
6       void iniciaAnimacion()
7       {
8         if (!ejecutandose)
9         {
10          setTouchable(false);
11          Application::getInstance()->registerTimerWidget(this);
12          ejecutandose=true;
13        }
14      }
15
16    protected:
17    . . .
18  }
```

Código 4.44: Definición del método iniciaAnimacion de la clase ElementoMenu

Este método inicia la animación del panel desplegable. Si el atributo interno *autodespliegue* es verdadero —su valor por defecto—, al pulsar el botón del componente, se llama a este método; si es falso —cuando el componente pertenece a un menú—, ha de ser el propio objeto que representa el menú el que lo llame cuando sea necesario. El método *iniciaAnimacion* —líneas de la 6 a la 14—, verifica, inicialmente, si no se está ejecutándose ya la animación —línea 8—, y si no lo está, impide que el componente sea sensible a pulsaciones —línea 10— y lo registra para poder gestionar el evento de temporización —línea 11— para poder llevar a cabo la animación. Finalmente marca que la animación ha comenzado —línea 12—.

En el archivo fuente, ElementoMenu.cpp, se definen los métodos con más código o que se consideran los más importantes. Estos métodos son: el constructor, el manipulador interno del evento *Click* del botón, *AlBotonMenu*, el método de asignación de menú, *ponMenu* y el manipulador del evento de temporización *handleTickEvent*.

La definición del constructor es la siguiente:

```
1  ElementoMenu::ElementoMenu():
2          Container(),
3          longitudDesplegado(0),
4          orientacion(HORIZONTAL),
5          ejecutandose(false),
6          retardo(0),
7          duracion(20),
8          Ecuacion(&EasingEquations::quadEaseOut),
9          id(-1),
10         autodespliegue(true),
11         menu(0),
12         botonMenuCallback(this, &ElementoMenu::AlBotonMenu)
13 {
14     botonMenu.setAction(botonMenuCallback);
15     contenido.add(imagenFondo);
16     Container::add(contenido);
17     Container::add(botonMenu);
18     Container::add(texto);
19     contenido.setVisible(false);
20 }
```

Código 4.45: Definición del constructor de la clase ElementoMenu

En la lista de inicialización del constructor, primeramente, se llama al constructor de la clase base: Container. Luego se inicializan los atributos internos de la clase. Así, como valores iniciales, se establece que la longitud del panel desplegable es cero, que la orientación del menú es horizontal... Al final de la lista de inicialización —línea 12—se enlaza el objeto de retrollamada de usuario — *botonMenuCallback* de tipo Callback— a un manipulador de usuario —*AlBotonMenu*— pero para utilización interna de la clase, y de esta forma poder transmitir el evento fuera de la clase mediante el puntero *MenuSeleccionadoCallback*. En el cuerpo del constructor se enlaza el objeto retrollamada *botonMenuCallback*, al puntero interno de tipo *GenericCallback* que tiene el botón del componente —línea 14—, de esta forma se enlaza el evento *Click* del botón, al exterior del componente. El resto del constructor se dedica a añadir cada uno de los componentes que constituyen el componente *ElementoMenu* —líneas de la 15 a la 18—, para finalmente establecer el panel desplegable —*contenido*— a invisible.

Las definiciones de los métodos AlBotonMenu y ponMenu son:

```
1  void ElementoMenu::AlBotonMenu(const AbstractButton& source)
2  {
3
4      if (autodespliegue && longitudDesplegado!=0)
5          iniciaAnimacion();
6
7      if (menu)
8          menu->gestionaEstados(id);
9
10     if (MenuSeleccionadoCallback && MenuSeleccionadoCallback->isValid())
11         MenuSeleccionadoCallback->execute(*this, botonMenu.getState());
12 }
13
14 void ElementoMenu::ponMenu(Menu* menu)
15 {
16     this->menu = menu;
17 }
```

Código 4.46: Definiciones de AlbotonMenu y ponMenu de la clase ElementoMenu

El método *AlBotonMenu* es el manipulador interno de usuario que gestiona el evento *Click* del botón y lo transmite al exterior de la clase. Primeramente inicia la animación llamando al método *inicializaAnimación* —ya comentado— solo si el atributo *autodespliegue* es verdadero —es verdadero por defecto a no ser que se añada el componente a algún menú— y el panel desplegable tenga longitud. Luego, si pertenece a un menú, llama a su método *gestionaEstados*, pasándole su identificador —que corresponde con el índice del array de tipo puntero a *ElementoMenu* que maneja el menú—, para que ejecute la máquina de estados del menú —ver siguiente sección—. Si se ha enlazado manipulador de usuario al puntero *MenuSeleccionadoCalback*, se transmite el evento a ese manipulador —se transmite el evento al exterior—. El método *ponMenu* —línea de la 14 a la 17— se dedica a cargar el puntero interno que tiene la clase *ElemetoMenu* con el objeto *Menu* que lo contiene. Este es un método que usará el objeto de tipo *Menu* cuando agregue un objeto *ElmentoMenu*.

Para manejar las animaciones del panel al desplegarse y replegarse, se implementa el siguiente manipulador de temporización de componente:

```
1  void ElementoMenu::handleTickEvent()
2  {
3   if (ejecutandose)
4   {
5    if (contador <= retardo) //gestión del retardo inicial
6    {
7      contador++;
8      contenido.setVisible(true);
9    }
10   else if (contador <= (uint32_t) (retardo + duracion)) //gestión de la animación
11   {
12     uint32_t contadorAnimacion = contador - retardo;
13     int16_t deltaDimension;
14     Container::invalidate();
15     if (botonMenu.getState()) //panel desplegándose
16     {
17       deltaDimension=(int16_t)Ecuacion(contadorAnimacion, 0, longitudDesplegado, duracion);
18       switch (orientacion)
19       {
20         case HORIZONTAL:
21              setWidth(botonMenu.getWidth()+deltaDimension);
22              contenido.setX(-longitudDesplegado+botonMenu.getWidth()+deltaDimension);
23              break;
24         case VERTICAL:
25              setHeight(botonMenu.getHeight()+deltaDimension);
26              contenido.setY(-longitudDesplegado+botonMenu.getHeight()+deltaDimension);
27              break;
28       }
29     }
30     else if (!botonMenu.getState())//panel replegándose
31     {
32       deltaDimension=(int16_t)Ecuacion(contadorAnimacion, 0, longitudDesplegado, duracion);
33       switch (orientacion)
34       {
35         case HORIZONTAL:
36              setWidth(botonMenu.getWidth()+longitudDesplegado-deltaDimension);
37              contenido.setX(botonMenu.getWidth()-deltaDimension);
38              break;
39         case VERTICAL:
40              setHeight(botonMenu.getHeight()+longitudDesplegado-deltaDimension);
41              contenido.setY(botonMenu.getHeight()-deltaDimension);
42              break;
43       }
44     }
45     Container::invalidate();
46     contador++;
47   }
48   else //gestión de la finalización de la animación
49   {
50     Application::getInstance()->unregisterTimerWidget(this);
51     ejecutandose = false;
52     contador = 0;
53     botonMenu.invalidate();
54     if (!menu || (menu && menu->obtenEstado()!=Menu::MENU_DESPLEGANDO))
55        setTouchable(true);
56     if (!botonMenu.getState())
57        contenido.setVisible(false);
58     if (AnimacionMenuTerminadaCallback && AnimacionMenuTerminadaCallback->isValid())
59         AnimacionMenuTerminadaCallback->execute(*this,botonMenu.getState());
60     if (menu && menu->obtenEstado()==Menu::MENU_DESPLEGANDO)
61        menu->gestionaEstados(id);
62   }
63  }
64 }
```

Código 4.47: Método handleTickEvent de la clase ElementoMenu

Hay que recordar que este manipulador de evento —*handleTickEvent*— solo entra en acción cuando el componente se ha registrado en el objeto de la clase *Application* para que ello suceda, y este registro acontece al ser llamado el método *iniciaAnimacion* —tanto desde el propio objeto de tipo *ElementoMenu*, cuando trabaja en solitario, como desde el objeto de tipo *Menu*, cuando trabaja de forma conjunta—. Está dividido en tres secciones. La inicial —líneas de la 5 a la 9—, que gestiona el retardo previo a la animación, que por defecto es de cero ticks; una intermedia —líneas de la 10 a la 47—, que gestiona la animación del despliegue y repliegue del panel y la final —líneas de la 48 a la 64—, que gestiona las acciones a llevar a cabo después de haber finalizado la

animación. Para realizar todo esto, la clase cuenta con el atributo *contador*, que será incrementado en cada *tick* —línea 7 y 46—, y será puesto a cero al terminar la animación —línea 52— para quedar en disposición de poder gestionar la siguiente animación. Para manejar el retardo, previo a la animación, se cuenta con el atributo *retardo* —línea 5—, que permite una espera de tantos *tick*, acumulados en el atributo contador, como indique *retardo*. Una vez finalizado el retardo, entra en juego el código que gestiona la animación. La duración de esta está controlada por el atributo *duracion*, pero el contador cuenta todos los ticks de la animación incluyendo el retardo inicial, por ello, se utiliza de forma interna, la variable *contadorAnimacion* —línea 12— que cuenta los *ticks* exclusivamente de la animación. La variable *deltaDimension* —línea 13—, indica en cada *tick*, cuál es el incremento de dimensión del panel desplegable para la animación. Para determinar este incremento se cuenta con la ecuación *quadEaseOut* de la clase *EasingEquation*s —ver sección 1.15—. Este incremento es positivo —líneas 21, 22, 25 y 26— cuando se debe ejecutar la animación de despliegue, o negativo —líneas 36, 37, 40 y 41— cuando la animación es de repliegue. La discriminación entre despliegue y repliegue está determinada por el estado del botón del elemento de menú; si está enclavado —línea 15—, se procede al despliegue, si se desenclava —línea 30—, se procede al repliegue. Para el despliegue y el repliegue, se modifica tanto el tamaño total del componente —líneas 21, 25, 36 y 40—, como la parte del panel visualizado en cada paso de animación —líneas 22, 26, 37 y 41—. La actualización de la visualización de la animación en cada paso, se asegura con la invalidación de todo el componente y esto se consigue con la invalidación de la clase base: *Container* —línea 45—. Ya que el elemento de menú puede estar dispuesto en forma horizontal o vertical, se ha de dar servicio a la animación de forma separada a cada situación. Cuando finaliza la animación, se ha de realizar ciertas tareas que permiten mantener la funcionalidad del componente. Lo primero es evitar que el componente siga gestionando el evento de temporización —línea 50—, indicar que ya no se está ejecutando la animación —línea 51— y poner a cero el contador de animación —línea 52—. Luego se procede a redibujar el botón del componente, para evitar no actualizaciones —línea 53—. Si el elemento de menú no pertenece a un menú o pertenece a uno y no se está desplegado el menú —no se están animando el menú para mostrar todos los elementos de menú—, se vuelve a permitir que el botón sea sensible a pulsaciones, que había sido insensibilizado al llamar al método *iniciaAnimación* del elemento del menú —líneas 54 y 55—. Si el panel ha sido replegado, se vuelve invisible el mismo —líneas 56 y 57—. Si se ha enlazado un manipulador de usuario para ser disparado al terminar la animación, se dispara —líneas 58 y 59—. Finalmente, si el elemento de menú pertenece a un menú y se ha pulsado sobre este elemento para desenclavarlo, se inicia la animación del todo el menú para mostrar los demás elemento de menú que habían sido ocultados al desplegar el panel de este elemento de menú —líneas 60 y 61—.

A continuación se presenta un ejemplo de declaración de dos elementos de menú en el archivo de cabecera de la vista:

```
1  class FftView : public View<FftPresenter>
2  {
3      public:
4          . . .
5      private:
6          . . .
7          //*******************************************************
8          //Declaración de elementos de menú
9          //*******************************************************
10         ElementoMenu elementoMenuVentanas;
11         ElementoMenu elementoMenuCursores;
12         . . .
13          //*******************************************************
14         //Declaración de controles del elemento de menú Ventanas
15         //*******************************************************
16         Widget widgetA, widgetB, . . .
17         //*******************************************************
18         //Declaración de controles del elemento de menú Cursores
19         //*******************************************************
20         Widget widgetL, widgetM, . . .
21         Container contCur1, contCur2;
22         SwipeContainer contCursores;
23 }
```

Código 4.48: Ejemplo de declaración de elementos de menú

Se declaran dos elementos de menú, *elementoMenuVentanas* y *elementoMenuCursores* —líneas 10 y 11, respectivamente—, donde se mostrarán dos configuraciones diferentes, con diferente complejidad. El primer elemento — *elementoMenuVentanas* —solo contendrá controles añadidos directamente a su panel desplegable; los controles *widgetA*, *widgetB*, . . ., de una hipotética clase Widget —aunque esta clase existe, no es posible crear objetos directamente desde ella, pero sirve como ejemplo de una clase inespecífica—. El segundo —*elementoMenuCursores*—, contendrá dos contenedores —*contCur1* y *contCur2*—, donde se añadirán los controles —controles hipotéticos *widgetL, widgetM*, . . .—. Estos dos contenedores serán a su vez añadidos a otro contenedor deslizante —*contCursores*, de tipo *SwipeContainer*—, para finalmente añadir este último al panel desplegable del elemento de menú. La manera como se hace, aparece en la siguiente porción de código perteneciente al código fuente de la vista:

```
1  void FftView::setupScreen()
2  {
3    //Configurar el elemento de menú Ventanas
4    elementoMenuVentanas.iniciializa(BITMAP_ELEMENTO_MENU_ID,BITMAP_ELEMENTO_MENU_SEL_ID);
5    elementoMenuVentanas.ponImagenFondo(BITMAP_FONDOMENU_ID);
6    elementoMenuVentanas.ponLongitudDesplegado(HAL::DISPLAY_WIDTH-\
7                                               (elementoMenu0.getWidth()));
8    elementoMenuVentanas.ponTexto(TypedText(T_RAPIDO));
9    //Añadir cada control al panel del elemento de menú Ventanas
10   elementoMenuVentanas.add(widgetA);
11   elementoMenuVentanas.add(widgetB);
12   . . .
13
14   //Configurar el elemento de menú Cursores
15   elementoMenuCursores.iniciializa(BITMAP_ELEMENTO_MENU_ID,BITMAP_ELEMENTO_MENU_SEL_ID);
16   elementoMenuCursores.ponImagenFondo(BITMAP_FONDOMENU_ID);
17   elementoMenuCursores.ponLongitudDesplegado(HAL::DISPLAY_WIDTH-\
18                                              (elementoMenuCursores.getWidth()));
19   elementoMenuCursores.ponTexto(TypedText(T_CURSORES));
20   elementoMenuCursores.setTerminado(MenuAnimacionTerminadaCallback);
21   //Añadir controles a cada contenedor
22   //y cada uno de ellos al panel del elemento de menú Cursores
23   contCur1.add(widgetL);
24   contCur2.add(widgetM);
25   . . .
26   contControles.add(contCur1);
27   contControles.add(contCur2);
28   . . .
29   elementoMenuCursores.add(contCursores);
30 }
```

Código 4.49: Ejemplo de definición de elementos de menú

Para el primer elemento de menú, inicialmente se procede a su configuración, especificando cuales son las dos imágenes de componen los estados presionado y liberado de su botón de selección —línea 4—; cuál será la imagen de fondo de su panel desplegable —línea 5—; cual será la longitud de este panel —líneas 6 y7—, y cuál será el texto que aparece en su botón de selección —línea 8—. Luego, se añaden cada uno de los controles que estarán contenidos en su panel desplegable —líneas 10 y 11—. Para el segundo elemento de menú se procede de igual forma en la configuración —líneas de la 15 a la 19—, con la salvedad que en este caso se programa un manipulador de evento de usuario cuando finalice la animación de despliegue o repliegue de este elemento de menú —línea 20—. A continuación, cada control será añadido a sendos contenedores, *conCur1* y *conCur2* —en las líneas respectivas 23 y 24—. Las posiciones relativas de estos controles, dentro de sus contenedores, han debido ser previamente establecidas —no mostrado en el código—. Luego, ambos contenedores son añadidos a un contenedor de tipo *SwipeContainer* —*contCursores*—, de este modo, el usuario podrá seleccionar un contenedor u otro, mediante el deslizamiento con el dedo sobre el panel desplegable del elemento de menú. Para que todo funcione, este ultimo contenedor —que contiene los dos contenedores, que a su vez contienen controles—, ha de ser añadido al panel desplegable del elemento de menú —línea 29–. Tal y como está configurado este último elemento de menú, al cambiar de un contenedor a otro —mediante deslizamiento con el dedo—, se seleccionarán los controles de un contenedor u otro, pero conservando la imagen de fondo del panel desplegable del elemento de menú. Es decir, cuando se deslice con el dedo, se apreciará como se deslizan los controles pero con el fondo inmóvil. Si se desea que además de los controles, se deslice con ellos un fondo, habrá

que añadir un fondo propio a cada contenedor —en *contCur1* y *contCur2*—. Para finalizar, estos dos elementos de menú han de estar integrados en un mismo menú, para ello se ha de crear un objeto de la clase *Menu* y añadir a este los dos elementos a este menú —ver siguiente sección para explicación de esta clase y ejemplos de utilización—.

4.4.1.10 Componente gráfico «Menu»

El componente *Menu* es el encargado de contener elementos de menú. Es por ello que deriva del componente *Container*, pero no se trata de un nuevo componente creado a partir de un contenedor a modo de composición de varios otros componentes como sí lo era el componente *ElementoMenu*. A pesar de derivar de una clase heredada de *Container*, *Menu* se trata de un nuevo componente creado por especialización de uno ya existente —ver sección 1.13.1—. Por ello, este componente no tiene una apariencia configurable sino que tiene la apariencia que tiene cada uno de los elementos de menú que lo componen. La declaración de la parte protegida de la clase, que se encuentra en el archivo de cabecera Menu.hpp, es la siguiente:

```
1  namespace touchgfx
2  {
3
4   class ElementoMenu; //declaración adelantada
5
6   class Menu : public ScrollableContainer
7   {
8      public:
9
10         Menu();
11         . . .
12     protected:
13
14          ElementoMenu *elementos[MAX_MENU];
15         uint8_t n_elementos;
16         int8_t idEnUso;
17         EstadosAnimacion estadosMenu;
18
19         Orientacion orientacion;
20         uint16_t inicio_menu;
21         uint8_t alpha;
22         bool animacionAlpha;
23         int16_t alfaInicial;
24         int16_t alfaFinal;
25
26          bool enEjecucion;
27         uint16_t contadorAnimacion;
28         uint16_t duracionAnimacion;
29         int16_t coord_inicial[MAX_MENU];
30         int16_t delta_coord;
31         EasingEquation EcuacionMov, EcuacionAlfa;
32
33         virtual void handleTickEvent();
34  };
35 }
```

Código 4.50: Declaración de la clase Menu

Está constituido por una serie de atributos que pueden ser divididos en cuatro grupos. El primero de ellos —líneas de la 14 a la 17—, está relacionado con los elementos que componen el menú. Así, el array de punteros *elementos*, alberga las direcciones de los objetos *ElementoMenu* que componen el menú; el máximo número de elementos en este array está fijado por la constante MAX_MENU definida en este mismo archivo de cabecera. El atributo *n_elementos*, contendrá el número real de elementos contenidos en este array, mientras que *idEnUso* representa el índice del array del elemento de menú que se está tratando. La variable *estadosMenu*, contiene el estado actual de la máquina de estados incluida internamente dentro de la clase. Esta variable es de tipo *EstadosAnimacion*, una enumeración definida dentro de la clase con los posibles valores de ESPERANDO_SELECCION, MENU_COLAPSANDO, ESPERANDO_DESELECCION, MENU_DESPLEGANDO. Más adelante se describirán estos estados junto con la máquina. En el segundo grupo de atributos están todos

aquellos relacionados con el aspecto. Así, se tiene el atributo *orientacion*, del tipo enumerado *Orientacion* —con los valores HORIZONTAL Y VERTICAL, definido dentro de la clase— que permite agrupar los elementos de menú en estas orientaciones. La variable *inicio_menu*, permite mover, de forma relativa todo el menú. Se accede a través del método *poneInicioMenu*. Luego están una serie de atributos relacionados con la transparencia. El atributo *alpha* establece la transparencia del menú, mientras que la variable booleana *animacionAlpha* posibilita que pueda existir un desvanecimiento del menú, cuando se selecciona un elemento concreto, o una reaparición, cuando el elemento es deseleccionado. Para ello cuenta con los atributos *alfaInicial* y *alfaFinal* que establecen los límites de transparencia para esta animación. En el tercer grupo están los atributos relacionados con la gestión de la animación. Así, *enEjecución* indica si la animación se está ejecutando —si tiene valor verdadero— o falso en caso contrario. El atributo *contadorAnimacion* acumula los *ticks* del sistema gráfico según se ejecuta la animación, mientras que en *duracionAnimacion* se establecen los *ticks* que durará la misma. Por otro lado, la variable *delta_coord* representa la distancia que a cada elemento le separa de la parte izquierda del menú —si la orientación es horizontal—, o de la parte superior del menú —si la orientación vertical—, con el objeto de poder crear la animación durante esta distancia cuando se selecciona un elemento. Para poder restituir los elementos a sus posiciones originales, cuando se deselecciona el elemento previamente seleccionado, se necesita guardar las posiciones iniciales de cada elemento. Para esto está el atributo *coord_inicial*, un array que guarda las posición de cada elemento de menú previo a la animación. Para gestionar la animación del movimiento del menú, al producirse una selección o deselección de un elemento, está la ecuación *EcuacionMov*, mientras que para tratar la animación alfa, se utiliza la ecuación *EcuacionAlfa*. Finalmente, en el último grupo, está exclusivamente el manipulador de evento de componente, que gestionará el evento de temporización: *handleTickEvent*.

Como este componente tiene que añadir varios elementos de menú, necesita un método que lo lleve a cabo:

```
1  void Menu::agrega(ElementoMenu& elemento)
2  {
3      if (n_elementos<MAX_MENU)
4      {
5          elemento.setXY(0,0);
6          elementos[n_elementos]=&elemento;
7          elemento.ponMenu(this);
8          elemento.id=n_elementos;
9          elemento.autodespliegue=false;
10         idEnUso=n_elementos;
11
12         if (n_elementos!=0)
13         {
14             for (int i=0; i<n_elementos; i++)
15             {
16                 if (i!=elemento.id)
17                 {
18                     if (orientacion==HORIZONTAL)
19                         elementos[i]->moveRelative(elemento.getWidth(),0);
20                     else
21                         elementos[i]->moveRelative(0,elemento.getHeight());
22                 }
23             }
24         }
25         add(elemento);
26         n_elementos++;
27     }
28 }
```

Código 4.51: Método agrega de la clase Menu

Se trata del método agrega, cuyo argumento es la referencia del objeto de tipo *ElementoMenu* a agregar. Esta agregación solo se realiza si no se sobrepasa el límite de elementos en el menú —línea 3—. Si se realiza la agregación, se modifica las coordenadas del elemento a añadir para ser el elemento más a la izquierda de los que componen el menú, si la orientación es horizontal, o más arriba, si la orientación es vertical. Luego, la dirección del elemento a añadir es tomada y almacenada en el array *elementos*, con el fin de poder modificar atributos de cada elemento añadido desde el objeto de tipo *Menu* —hay que recordar que la clase *Menu* es una clase amiga de la clase *ElementoMenu* y por tanto, tiene acceso a las partes privadas y protegidas de esta última—. A

continuación, se envía al elemento añadido, la dirección del menú que lo contiene, con el fin que el elemento añadido pueda interactuar con la máquina de estados que representa la clase *Menu*. Seguidamente, se asigna un identificador al atributo *id* del elemento añadido, que corresponde con el índice del array que contiene su dirección. Este identificador tiene el cometido de poder identificar al elemento de menú que interactúa con la máquina de estados en el objeto de tipo *Menu*. También se modifica el atributo *autodespliegue* del elemento añadido, para que no se inicie la animación de este al ser pulsado su botón; será la máquina de estados del objeto *Menu* la que inicie la animación del elemento cuando estime oportuno. El objeto *Menu* tiene el atributo *idEnUso*, que alberga el valor del índice del elemento de menú que se está tratando, por ello, al añadir un nuevo elemento de menú, este atributo adquiere el valor de su identificador *id*. El objetivo final de este método es añadir el nuevo elemento al contendor que representa el objeto de tipo *Menu*, ya que desciende de *Container*, por eso, cada elemento pasado a esta función es añadido mediante el método *add* —línea 25—. Pero se necesita un orden espacial para realizar esta adición, es por ello que se utiliza el bloque *for* entre las líneas 14 a 23. En este bloque, se mueven todos los elementos, ya existentes, hacia la derecha para poder añadir el nuevo en la zona más a la izquierda —si la orientación es horizontal—, o mover todos los componentes hacia abajo para poder posicionar el nuevo añadido en la zona de arriba —si la orientación es vertical—. Finalmente, el número de elementos, *n_elemntos*, es incrementado para reflejar la cantidad de elementos en el menú y para asignar este valor como identificador del futuro elemento a añadir y que corresponda con el índice del array que alberga su dirección —el array es *elementos*—.

La clase *Menu* implementa el manipulador de temporización para poder realizar animaciones de todo el menú —que no tienen que ver con las animaciones individuales de cada elemento, aunque sí deben estar sincronizadas—. Para esta sincronización, se utiliza una máquina que, se puede decir, está contenida en la clase *Menu* o que ella misma es la máquina de estados. Sin embargo no solo la clase *Menu* es la máquina de estados; interviene, también, la gestión de temporización de cada elemento de menú. Por tanto, la máquina de estados está constituida por el manipulador de evento de temporización de la clase *Menu*, por el manipulador de eventos de temporización de cada elemento de menú y por la gestión de cambio de estados de esta máquina, realizada en el método *gestionaEstados* de la clase *Menu*, que es utilizado por cada uno de los elementos de menú, así como por el propio menú. El esquema de la máquina de estados es la siguiente:

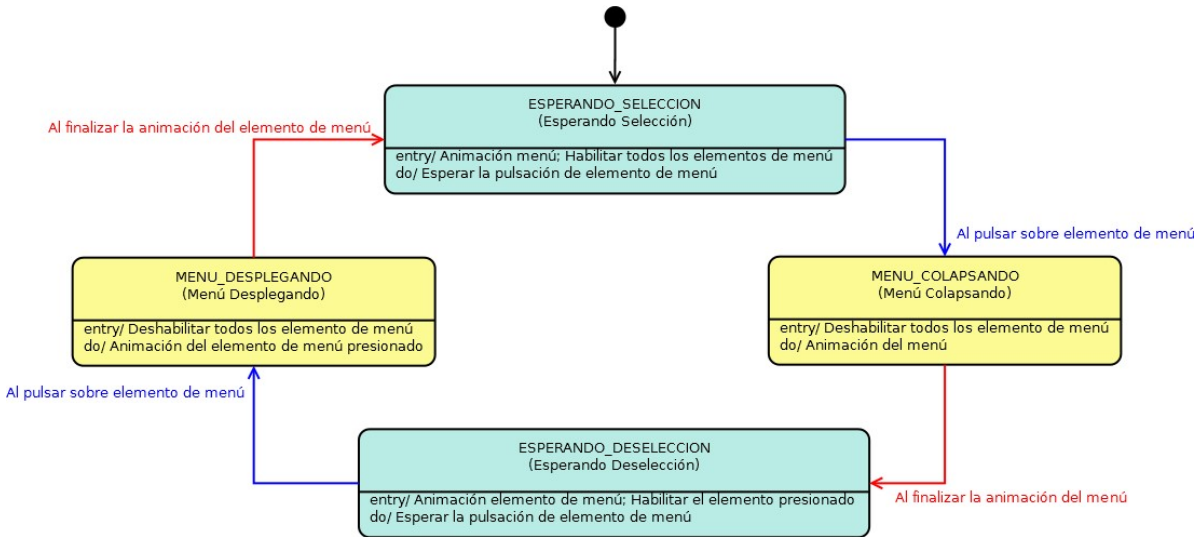


Ilustración 4.20: Máquina de estados de la clase Menu

Posee cuatro estados, dos de ellos estables —ESPERANDO_SELECCION y ESPERANDO_DESELECCION—, marcados en color azul claro, y dos inestables —MENU_COLAPSANDO y MENU_DESPLEGANDO—, marcados en amarillo. En los estados estables, la máquina permanece en ellos de forma indefinida a no ser que suceda la pulsación sobre un elemento de menú, lo que hace que se pase al siguiente estado. Así, el estado ESPERANDO_SELECCION, indica que el menú está mostrando todos los elementos de menú y se está a la espera de la pulsación en alguno de ellos para saltar al siguiente estado. En el caso del estado ESPERANDO_DESELECCION, indica que solo se está mostrando el elemento de menú, con su panel desplegado, que previamente ha sido seleccionado, y se está a la espera de volver a recibir la pulsación sobre él para pasar al siguiente estado. Los estados inestables están entre los dos estables para crear animaciones en el paso de uno a otro. Es por ello que se les cataloga de inestables. La permanencia en estos dos estados, coincide con la duración de la animación y al finalizar esta, se salta al siguiente estado: el estado estable correspondiente. Con ayuda del esquema anterior, se puede viajar, a través de los distintos estados de la máquina, en una situación en la que se parte del estado estable ESPERANDO_SELECCION como estado inicial. Como ha sido comentado, en este estado se muestran todos los elementos que componen el menú y todos ellos están habilitados. Cuando se produce una pulsación sobre alguno de estos elementos, para su selección, la máquina pasa al siguiente estado —MENU_COLAPSANDO—. Al entrar en este estado se deshabilitan todos los elementos de menú, ya que se va a producir la animación en la que todos los elementos se desplazan para dejar el elemento seleccionado en la zona más a la izquierda del menú, si su orientación es horizontal, o en la zona más hacia arriba, si la orientación es vertical, y no se puede recibir ningún evento de pulsación nuevo mientras tanto. Luego, se procede a realizar esta animación de todo el menú, donde, al finalizar, se dispara la transición de la máquina de estados a su siguiente estado: el estado estable ESPERANDO_DESELECCION. La primera acción que se toma en este, es ejecutar la animación del elemento seleccionado, quedando únicamente visible este elemento de menú con su panel desplegado y habilitado —ya que había sido deshabilitado junto con el resto de elementos del menú—. En este estado se permanecerá de forma indefinida, para que el usuario pueda interactuar con los controles contenidos en el panel desplegable de este elemento de menú, hasta que se pulse sobre su botón de selección para deseleccionarlo. Esto dispara la transición de la máquina al siguiente estado: el estado inestable MENU_DESPLEGANDO. Al entrar en él, lo primero que se realiza es deshabilitar todos los elementos de menú para que no sean sensibles a pulsaciones mientras la animación de todo el menú está en curso. Luego, se procede a realizar la animación del elemento de menú, en donde se esconde su panel —que contiene los controles relativos a este elemento de menú—. Cuando finaliza la animación del elemento de menú deseleccionado, se pasa al siguiente estado: el estado estable ESPERANDO_SELECCION. La acción que se realiza al entrar en él es ejecutar la animación del todo el menú, colocando cada elemento del mismo en la posición correcta, para a continuación, volver a habilitar todos los elementos de menú, quedando en disposición de ser nuevamente seleccionados. Con esto se ha completado una vuelta completa sobre todos los estados de la máquina. Como se muestra en la figura, el punto de entrada se realiza en el estado ESPERANDO_SELECCION.

La transición de estados se realiza llamando al método *gestionaEstados* del objeto *Menu*:

```
1  void Menu::gestionaEstados(int8_t id)
2  {
3      idEnUso=id;
4      switch (estadosMenu)
5      {
6          case ESPERANDO_SELECCION:
7              estadosMenu=MENU_COLAPSANDO;
8              for (int i=0; i<n_elementos; i++)
9                  elementos[i]->setTouchable(false);
10             if (orientacion==HORIZONTAL)
11                 enableHorizontalScroll(false);
12             else
13                 enableVerticalScroll(false);
14             Application::getInstance()->registerTimerWidget(this);
15             enEjecucion=true;
16             break;
17
18         case MENU_COLAPSANDO:
19             estadosMenu=ESPERANDO_DESSELECCION;
20             elementos[idEnUso]->iniciaAnimacion();
21             break;
22
23         case ESPERANDO_DESSELECCION:
24             estadosMenu=MENU_DESPLEGANDO;
25             for (int i=0; i<n_elementos; i++)
26                 elementos[i]->setTouchable(false);
27             elementos[idEnUso]->iniciaAnimacion();
28             break;
29
30         case MENU_DESPLEGANDO:
31             estadosMenu=ESPERANDO_SELECCION;
32             enEjecucion=true;
33             if (orientacion==HORIZONTAL)
34                 enableHorizontalScroll(true);
35             else
36                 enableVerticalScroll(true);
37             Application::getInstance()->registerTimerWidget(this);
38             break;
39     }
40 }
```

Código 4.52: Gestión de la transición de estados de la clase Menu

Básicamente, el párrafo anterior explica el comportamiento de esta función. Cuando es llamada por el elemento de menú, mediante el puntero al objeto de tipo *Menu*, que internamente tiene todo elemento de menú, pasa como argumento su identificador, que corresponde con el índice del array de punteros a objetos de tipo *ElemenoMenu*, que la clase *Menu* tiene. De esta forma, desde el objeto de tipo Menu, se pude modificar los atributos de cada elemento de menú, como pueda ser su habilitación o inicio de su animación —hay que recordar que la clase *Menu* es amiga de la clase *ElementoMenu*, y por tanto, la primera puede tener acceso a los atributos privados y protegidos de la segunda—. Para tener un acceso más sencillo al elemento de menú que llamó a este método, en el atributo *idEnUso* —de la clase Menu— se almacena el identificar *id* del elemento de menú que provoca la transición de la máquina, y que viene en el argumento. El elemento de menú va a provocar la transición de la máquina cuando esté previamente no seleccionado y acabe su animación, es decir, cuando la máquina esté en el estado MENU_DESPLEGANDO —mirar la anterior sección dedicada a la clase *ElementoMenu*—. También ejecutara la transición —este método— cuando se produzca una pulsación sobre su botón de selección, para seleccionar o deseleccionar el elemento en cuestión, es decir, se esté en los estados ESPERANDO_SELECCION o ESPERANDO_DESELECCION. Cuando es el propio objeto de tipo *Menu* el que llama a este método para la ejecución de la transición de estados, envía como argumento el identificador del elemento de menú que previamente se había seleccionado y la llamada a la transición se produce al finalizar la animación de todo el menú, es decir se está en el estado MENU_COLAPSANDO. Todo lo expuesto puede comprenderse más fácilmente si se revisa el esquema de la máquina de estados que aparece más arriba. Para la comprensión de este método, que gestiona las transiciones, hay que tener en cuenta que las cláusula *Case*, no indican al estado hacia el que se va en la transición, sino desde el que se parte. Para saber cuál es el estado hacia el que se transita, basta con mirar la sentencia que aparece en la siguiente línea del *Case*, en donde es asignada la variable *estadosMenu*.

Para gestionar la animación de todo el menú, así como ciertas acciones a tomar en la máquina de estados, se necesita que el objeto de la clase *Menu* gestione el evento de temporización, para ello implementa el manipulador *handleTickEvent*:

```
void Menu::handleTickEvent()
{
    if (!enEjecucion)
        ScrollableContainer::handleTickEvent();

    if (enEjecucion)
    {
        if (contadorAnimacion==0)
        {
            if (estadosMenu==MENU_COLAPSANDO)
            {
                if (orientacion==HORIZONTAL)
                    delta_coord=elementos[idEnUso]->getX();
                else
                    delta_coord=elementos[idEnUso]->getY();
                alfaInicial=255; alfaFinal=0;
            }
            else if (estadosMenu==ESPERANDO_SELECCION)
            {
                if (orientacion==HORIZONTAL)
                    delta_coord=-(coord_inicial[idEnUso]-elementos[idEnUso]->getX());
                else
                    delta_coord=-(coord_inicial[idEnUso]-elementos[idEnUso]->getY());
                alfaInicial=0; alfaFinal=255;
            }
            for (int i=0; i<n_elementos; i++)
            {
                if (orientacion==HORIZONTAL)
                    coord_inicial[i]=elementos[i]->getX();
                else
                    coord_inicial[i]=elementos[i]->getY();
            }
        }
        if (contadorAnimacion <= (uint32_t) (duracionAnimacion))
        {
            int16_t incremento_coord = (int16_t) EcuacionMov(contadorAnimacion, 0,\
                                                    delta_coord, duracionAnimacion);
            for (int i=0; i<n_elementos; i++)
            {
                elementos[i]->invalidate();
                if (orientacion==HORIZONTAL)
                    elementos[i]->setX(coord_inicial[i]-incremento_coord);
                else
                    elementos[i]->setY(coord_inicial[i]-incremento_coord);
                elementos[i]->invalidate();
            }
            if (animacionAlpha)
            {
                uint8_t incrementoAlfa=(uint8_t) EcuacionAlfa(contadorAnimacion, 0,\
                                                alfaFinal-alfaInicial, duracionAnimacion);
                for (int i=0; i<n_elementos; i++)
                {
                    if (i!=idEnUso)
                        elementos[i]->setAlpha(alfaInicial + incrementoAlfa);
                }
            }
            contadorAnimacion++;
        }
        else
        {
            contadorAnimacion  = 0;
            Application::getInstance()->unregisterTimerWidget(this);
            enEjecucion=false;
            if (estadosMenu==ESPERANDO_SELECCION)
                for (int i=0; i<n_elementos; i++)
                    elementos[i]->setTouchable(true);
            if (estadosMenu==MENU_COLAPSANDO)
                gestionaEstados(idEnUso);
        }
    }
}
```

Código 4.53: Manipulador handleTickEvent de la clase Menu

La gestión de este manipulador se puede dividir en cuatro partes. En la primera, la manipulación del evento de temporización se delega en la clase antecesora, si no se está ejecutando la animación del menú —líneas 3 y 4—. Esto es debido a que la clase antecesora —*ScrollableContainer*— necesita manejar temporizaciones para su normal funcionamiento. En la segunda parte, se preparan diversos atributos para poder iniciar la animación —líneas de la 8 a la 33—. Así, el atributo *delta_coord* almacena la distancia que separa al elemento de menú seleccionado o deseleccionado de su posición final en la animación. Esta distancia es positiva, si se ha seleccionado el elemento, o negativa si se ha deseleccionado. Esta será la distancia que se desplazará todo el menú en la animación, por lo que es necesario recordar las posiciones originales de los elementos de menú antes de que esta se ejecute. Esto se almacena en el array *coord_inicial*, donde en cada elemento del mismo se almacenan las respectivas coordenadas iniciales de cada elemento de menú. En el almacenamiento de los valores de estos dos atributos, se ha de tener en cuenta la orientación del menú. Si también se configura la animación para que tenga apagado y encendido alfa —mediante el atributo booleano *animacionAlpha*—, se han de establecer los valores iniciales y finales alfa para cada estado de animación —desplegando menú o entrando en el estado de espera de selección—. En la tercera parte —líneas de la 34 a la 58—, se realiza la animación del menú acorde con los atributos que se configuraron en la parte previa. La duración de la animación está fijada por el atributo *duracionAnimacion*, mientras que el que establece el paso de animación es *contadorAnimacion*. Con estos dos atributos y el que fija el desplazamiento de todo el menú — *delta_coord* —, se calcula, a través de una ecuación de tipo *EasingEquation* —*EcuacionMov*—, el incremento de desplazamiento —almacenado en el atributo *incremento_coord*— para cada paso de temporización, que como se ha mencionado, está presente en el atributo *contadorAnimacion*. Si se ha habilitado la animación alfa, se realiza a través de la ecuación *EcuacionAlfa*. En la última parte —líneas de la 59 a la 69— se tramita la finalización de la animación, donde se reinicia el contador de animación —línea 61—, se impide que se gestione más eventos de temporización —línea 62—, se señaliza que no se está ejecutando la animación —línea 63—, si se está en el estado de espera de pulsación, se habilitan todos los elemento de menú y si se está en el estado de colapso de menú, se ejecuta la transición al estado de espera de deseleccionar el elemento de menú previamente seleccionado.

A pesar de la posible complejidad de funcionamiento de esta clase, su utilización es muy sencilla. Basta con declarar un objeto de tipo Menu en la cabecera de la vista:

```
1  . . .
2  #include <gui/common/ElementoMenu.hpp>
3  #include <gui/common/Menu.hpp>
4  . . .
5  class FftView : public View<FftPresenter>
6  {
7  public:
8      . . .
9  private:
10     //*********************************************************************
11     //Declaración del menú principal
12     //*********************************************************************
13     Menu menu1;
14     . . .
15 };
```

Código 4.54: Ejemplo de declaración de objeto de tipo Menu

Y configurarlo en el método setupScreen de la vista en su fichero fuente:

```
1  void FftView::setupScreen()
2  {
3    . . .
4    //********************************************************************************
5    //Definición del menú principal
6    //********************************************************************************
7    menu1.setPosition(0,182,HAL::DISPLAY_WIDTH, HAL::DISPLAY_HEIGHT/3);
8
9    //********************************************************************************
10   //Agregación de controles al menú principal y configuración del mismo
11   //********************************************************************************
12   menu1.agrega(elementoMenu0);
13   menu1.agrega(elementoMenu1);
14   menu1.ponerInicioMenu((menu1.getWidth()-
15                          (menu1.obtenN_Elementos())*elementoMenu0.getWidth())/2);
16   menu1.ponerAnimacionAlpha(true);
17
18   //********************************************************************************
19   //Agregación del objeto de tipo Menu la vista
20   //********************************************************************************
21   add(menu1);
22   . . .
23 }
```

Código 4.55: Ejemplo de definición y configuración del objeto de tipo Menu

En este caso, se configura el menú para que tenga una posición concreta, utilizando el método *setPosition* que todos los controles poseen. Luego se añaden los elementos de menú *elementoMenu0* y *elementoMenu1*, se centra todo el menú utilizando su método *ponerInicioMenu* y se habilita la animación alfa —tal y como se hace en el código del proyecto—. Para fina

4.4.1.11 Componentes gráficos «Graph» y «Serie».

Al inicio de este proyecto, la librería *TouchGFX* no disponía de ningún componente que representara una gráfica en pantalla. Hacia la mitad del mismo, el fabricante de la librería desarrolló uno que puso a libre disposición mediante un repositorio. Este es un componente que deriva de *CanvasWidget* y utiliza datos fraccionarios en formato de coma fija. Pero la necesidad de tener, desde el inicio, un componente que realizara gráficas sobre pantalla —o al menos un esqueleto del mismo en el que apoyarse—, obligaba a desarrollar uno propio. Esta idea se vio reforzada por el hecho de tener que tratar la posibilidad de representar 1024 puntos en pantalla refrescados de forma periódica, ya que el componente que finalmente suministra *TouchGFX* muestra una veintena y de forma permanente —al menos en los códigos de ejemplo—. El componente para realizar gráficas de desarrollo propio es el conjunto *Graph-Serie*, que deriva de la clase *Widget*, y por tanto su elaboración parte «desde cero». Su aspecto en este proyecto es el siguiente:

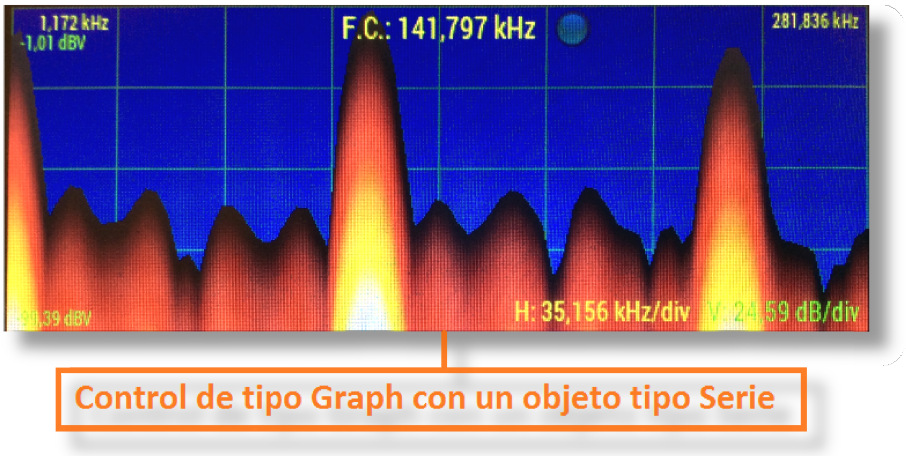


Ilustración 4.21: Apariencia del control Graph-Serie en este proyecto

Este es uno de los componentes, sino el más, importante de todo el proyecto. Su misión es la representación gráfica de datos en pantalla. La responsabilidad de esta representación estará repartida entre dos clases: *Graph* y *Serie*. La función básica de la clase «*Serie*» es la de encapsular una representación gráfica y todas aquellas características y atributos propias de la serie gráfica, mientras que la de la clase *Graph* es contener una o varias *Series* y pintar en pantalla cada una de ellas. La relación entre *Graph* y *Serie* es, por un lado, de dependencia, ya que *Graph* no puede realizar su trabajo sin un objeto de la clase *Serie*, y por otro lado de agregación, ya que un objeto de la clase *Graph* contiene uno o muchos objetos de la clase *Serie*. Para ello, la clase *Graph* dispone de un *array* de punteros a objetos de la clase *Serie*. Debido a esta fuerte relación, se decide definir la clase Serie dentro de la clase *Graph*, con lo que se puede hablar del componente *Graph-Serie*:

```
1  class Graph : public Widget
2  {
3      public:
4
5          Graph();// constructor
6          virtual void draw(const Rect& invalidatedArea) const;
7          virtual Rect getSolidRect() const;
8
9   //////// Inclusión de la definición de la clase "Serie" /////////
10         class Serie
11         {
12
13             public:
14
15                 Serie():// constructor
16
17             . . .
18
19         };
20 /////////// Fin de la definición de la clase "Serie" ////////////
21
22     . . .
23
24 };
```

Código 4.56: Definición de la clase «Serie»

El componente costa de dos ficheros, uno el de cabecera —Graph.hpp—, donde se encuentran las declaraciones de las clases *Graph* y *Serie*, así como la de varias constantes, y otro, el archivo fuente —Graph.cpp— donde se desarrolla el código de cada uno de los métodos.

El paradigma de gráfica y serie es utilizado por varios programas de representación gráfica, en concreto en Microsoft *Excel*, donde una gráfica se incrusta en el documento y es utilizado para contener una o varias representaciones de datos —las series—. El desarrollo de este componente sigue la misma filosofía y está preparado para contener cuatro series a la vez, aunque esto puede ser modificado variando el valor de la constante *MAXSERIES* definida en el archivo de cabecera:

```
1  . . .
2
3  #include <stdlib.h>
4
5  using namespace touchgfx;
6
7  . . .
8
9  #define MAXSERIES 4
10
11 #include <gui/common/Cursor.hpp>
12 class Graph : public Widget
13 {
14
15   . . .
16
17     public:
18       void agregaSerie(Serie &serie, . . .)
19         { . . . }
20
21     private:
22
23         uint8_t nSeries;
24         Serie *Series[MAXSERIES];
25
26   . . .
27
28 };
```

Código 4.57: Definición del número de series en el gráfico

En este fragmento de código se muestra la constante *MAXSERIES* definida con el valor 4. Está constante determina el valor máximo de series a representar. Si se desea representar más series en el gráfico, solo hay que modificar el valor de este «define», teniendo en cuenta que a más series representadas, mayor carga de trabajo para el microcontrolador y mayor tiempo necesario para la representación. En cualquier caso, la carga de trabajo vendrá dada por el número de series utilizadas en la representación y no por las disponibles. Por ejemplo, en el código anterior están disponibles como máximo cuatro series, pero si solo se utiliza una, la carga será únicamente la necesaria para representar esta serie. Este «define» determina el número de punteros a objetos de tipo «*Serie*» del *array Series* —línea 24—, *array* que utiliza la clase *Graph* para manejar las series disponibles. También aparece el atributo *nSeries* —línea 23— que contiene el número de series realmente contenidas en el *array*. Este atributo será actualizado en cada agregación de una serie al gráfico —mediante el procedimiento *agregaSerie*, mostrado en la línea 18— y es utilizado dentro el método *draw* de la clase *Graph* para manejar las series disponibles. Se recuerda que el método *draw* está presente en todas las clases derivadas de *Widget* —línea 12— y que han de implementar para dibujar el componente —sección 1.13.3—.

La idea de manejar varias representaciones gráficas a la vez era debido a que inicialmente se pensó en desarrollar un aplicación de tipo osciloscopio con dos trazas, pero finalmente se decantó por una aplicación que mostrara la transformada de Fourier de datos temporales adquiridos en los *ADCs* del micro, con lo que finalmente solo se utiliza un objeto de tipo *Serie*.

En la línea 11 aparece la inclusión del archivo de cabecera «Cursor.hpp» que contiene la definición de la clase que representa el cursor de medida. La clase *Graph* dispone de otro *array* de punteros a objetos de la clase «Cursor», pero la diferencia fundamental con respecto a la clase «Serie» es que la primera está definida en un archivo aparte mientras que la segunda, como se ha visto, lo está dentro de la clase *Graph*. Ello es debido a la no existencia de una fuerte relación entre las clases *Graph* y *Cursor*, ya que un objeto de la clase *Graph* puede realizar la representación de una gráfica sin necesidad de tener cursores. De hecho, la inclusión del código necesario para poder gestionar los cursores se realizo con posterioridad a la creación de la clase *Graph*, en el momento del desarrollo de este proyecto en el que se decidió su uso. Las características concretas de la clase *Cursor*, su relación con la clase *Graph* y modo de empleo serán comentadas en la sección 4.4.1.12.

4.4.1.11.1 Origen de datos de las series.

Los datos de la gráfica no están contenidos en ningún objeto, sino que se encuentran en una región de memoria donde la tarea de la aplicación, encargada de adquirirlos y tratarlos, los almacena —se trata de un *array* reservado estáticamente de longitud 4096 bytes—. Ya que esta región es un recurso compartido entre el productor –la tarea encargada de la adquisición y cálculo de la FFT —y el consumidor la tarea del entorno gráfico—, se hace necesario un sistema de arbitración consistente en una cola del sistema de tiempo real *FreeRTOS*, llamada *colaDeAplicacion_a_GUI*, definida en la función *main*, que sirve de enlace entre productor y consumidor a través de envío de mensajes, en cuyo contenido está, entre otra información, la región de memoria a tratar. Todo este mecanismo es explicado en mayor detalle en las secciones 4.5.10 y 4.6.3. La cuestión a tratar aquí, después de tener permiso para acceder a la región de memoria, es como se interpretan los datos contenidos en la misma.

Un atributo esencial es un puntero hacia esta región de memoria. Ya que la clase encargada de encapsular la representación gráfica es la clase *Serie*, este puntero debe ser un atributo de la misma y su nombre dado será *Datos*. Para poder cargar este puntero con la dirección de la memoria se provee de un método público llamado *PonDatos*, cuyo primer argumento es la dirección de la región y el segundo es el número de datos contenidos. El tipo de datos almacenados en esta región debe ser versátil, con lo que el tipo del puntero «Datos» será *void*, permitiendo que se apunte a datos de diverso tipo previa conversión explícita —*cast*—. Está conversión está facilitada por el método público *PonTipoDato*, de la clase *Serie*, cuyo único argumento es una enumeración con los posibles tipos aceptados. Esta enumeración está definida internamente en la clase *Serie* con acceso público y se llama *TipoDato*, cuyos posibles valores son *PUNTERO_8BITS*, *PUNTERO_16BITS* y *PUNTERO_32BITS*, para especificar que el puntero *Datos* accede a una región de datos de tipo byte, palabra o palabra doble, respectivamente. Para facilitar la carga del puntero, a través del método *PonDatos*, no teniendo que especificar el número de datos, se provee de unas macros —definidas en el archivo de cabecera de la clase *Graph-Serie*— con el nombre-formato *CARGA_ARRAY_XBITS(x)*, donde «XBITS» hace referencia al tipo de datos contenidos en la región de memoria y que pueden tener el valor de «8BITS», «16BITS» o «32BITS». Para aclarar lo expuesto se presenta, a continuación, la parte de la clase «Serie» involucrada es todas estas cuestiones:

```
1  #define CARGA_ARRAY_8BITS(x) x,sizeof(x)/sizeof(uint8_t)
2  #define CARGA_ARRAY_16BITS(x) x,sizeof(x)/sizeof(uint16_t)
3  #define CARGA_ARRAY_32BITS(x) x,sizeof(x)/sizeof(uint32_t)
4
5  class Graph : public Widget
6  {
7   . . .
8   class Serie
9   {
10   public:
11
12      . . .
13      typedef enum
14       {
15          PUNTERO_8BITS,
16          PUNTERO_16BITS
17          //PUNTERO_32BITS,
18       }TipoDato;
19
20      void ponDatos(void* Datos, uint16_t nDatos)
21          {this->Datos=Datos; this->nDatos=nDatos; . . . }
22
23    protected:
24
25      . . .
26      //atributos COMPORTAMIENTO
27      TipoDato tipoDato;
28      void* Datos;
29      uint16_t nDatos;
30      . . .
31   };
32   . . .
33 };
```

Código 4.58: Parte de la clase «Serie» dedicada a la carga de datos

En la línea 28 está definido el puntero a los datos —*Datos*—, y en la 29 el número de los datos —*nDatos*—. Estos dos atributos son asignados mediante la llamada al método definido y declarado entre las líneas 20 a 21 —*ponDatos*—. El tipo de dato apuntado se especifica mediante el atributo *tipoDato* definido en la línea 27, cuyos únicos valores que puede albergar están definidos en la enumeración *TipoDato* —líneas 13 a 18—. Finalmente, están la macros, definidas en las líneas 1 a 3, que sirven para utilizar un único argumento en la llamada al método *ponDatos*. Por ejemplo, teniendo un *array* de tipo 16 bits con 2048 datos llamado *fuenteDatos*, y un objeto de tipo *Serie*, llamado *serie1*, el código utilizado para la carga de datos sería el siguiente:

```
1  . . .
2
3  uint8_t fuenteDatos[4096];
4  Graph::Serie serie1;
5
6  serie1.ponTipoDato(Graph::Serie::PUNTERO 16BITS);
7  serie1.ponDatos(CARGA ARRAY 16BITS(fuenteDatos));
8  // equivale a: serie1.ponDatos(fuenteDatos, 2048);
9  . . .
```

Código 4.59: Carga de datos en el objeto de tipo «Serie»

De este código hay que destacar varios conceptos no explicados anteriormente. En la línea 1 se ha declarado el *array* de datos con tipo de 8 bits y de 4096 datos, cuando anteriormente se ha dicho que eran 2048 datos de 16 bits. Se ha hecho así intencionadamente para mostrar que los datos los tomaremos de la longitud que queramos (8, 16 o 32 bits); esto solo dependerá del tipo de puntero que acceda a los mismos. La única restricción es mantener la cantidad de datos tratados de forma coherente al valor inicialmente reservado para ellos y el modo de acceso a los mismos, de otra forma se pueden producir desbordamientos y colapsar todo el sistema. Así 4096 datos de 8 bits son 2048 datos de 16 bits, ya que cada dato de 16 bits es el doble de longitud que el de 8 bits. Accediendo a los datos como paquetes de 16 bits, el índice máximo del *array* sería de 2047. En el caso de acceder a los datos como paquetes de 32 bits, serían 1024 datos y el índice máximo de 1023. Otro punto a destacar es la forma de definición del objeto *serie1* —línea 4— y la especificación del valor del tipo de dato —línea 6—. En ambos caso se utiliza la especificación «Graph::Serie». Para el caso de la declaración del objeto *serie1*, se utiliza esta especificación debido a que la clase «Serie» está definida dentro de la clase *Graph* —como ya ha sido comentado—. En una enumeración definida dentro de una clase, para usar alguno de sus posibles valores, se utiliza igualmente la especificación de la clase donde está contenido seguida del operador de resolución de alcance para finalmente obtener el valor. En el caso concreto mostrado en la línea 6 se utilizan dos especificaciones de clase con sus correspondientes operadores de resolución de alcance, debido a que el tipo enumerado está dentro de la clase *Serie* que a su vez está dentro de la clase *Graph*. Lo último a destacar es la existencia de las macros que se usan como argumento del método *ponDatos*. Su utilidad es doble, por un lado solo se usará un argumento: el nombre del *array* que contiene los datos. Por otro, se asegura el valor correcto de los datos a tratar. La única precaución a tener es usar la macro acorde con el tipo de datos a acceder.

Se ha explicado hasta ahora el código relacionado con la carga de datos, pero no el relacionado con el acceso a ellos. Para esto se utiliza la sobrecarga del operador indexación[19] y dentro del mismo se tiene en cuenta el valor del atributo *tipoDato*:

[19] Operador de acceso a elemento de array: [ ]

```
1  #define CARGA_ARRAY_ALTERNADO_8BITS(x,desplazamiento)\
2                                          (x+desplazamiento),sizeof(x)/sizeof(uint8_t)/2
3  #define CARGA_ARRAY_ALTERNADO_16BITS(x,desplazamiento)\
4                                         (x+desplazamiento*2),sizeof(x)/sizeof(uint16_t)/2
5  #define CARGA ARRAY ALTERNADO 32BITS(x,desplazamiento)\
6                                         (x+desplazamiento*4),sizeof(x)/sizeof(uint32_t)/2
7  . . .
8  class Graph: public Widget
9  {
10  . . .
11  class Serie
12  {
13   public:
14     . . .
15     void poner_Alternada(bool a) {SerieAlternada=a;}
16     bool Es_Alternada(void) {return SerieAlternada;}
17     inline uint32_t operator[](int i)
18       {
19         uint16_t retorno=0;
20         if (tipoDato==PUNTERO_8BITS)
21            retorno=(SerieAlternada==false)?((uint8_t*)Datos)[i]:((uint8_t*)Datos)[i<<1];
22         else if (tipoDato==PUNTERO_16BITS)
23             retorno=(SerieAlternada==false)?((uint16_t*)Datos)[i]:((uint16_t*)Datos)[i<<1];
24         else if (tipoDato==PUNTERO_32BITS)
25             retorno=(SerieAlternada==false)?((uint32_t*)Datos)[i]:((uint32_t*)Datos)[i<<1];
26         return retorno;
27       }
28     . . .
29   protected:
30     . . .
31     //atributos COMPORTAMIENTO
32     TipoDato tipoDato;
33     bool SerieAlternada;
34     void* Datos;
35     uint16_t nDatos;
36  };
37  . . .
38 };
```

Código 4.60: Sobrecarga del operador indexación de la clase «Serie»

Un concepto nuevo que se maneja en esta parte del código de la clase «Serie» es la «serie alternada». Cuando un objeto de tipo «Serie» se marca como «serie alternada», se está especificando que los datos que maneja dicho objeto realmente constituyen dos grupos de datos, donde los dato de uno de los grupos ocupan las posiciones pares del *array* y el otro las impares. Esta nueva forma de trabajar viene impuesta por los periféricos ADC del micro STM32F4 cuando trabaja en modo múltiple simultáneo[20], entregando sus datos mediante DMA de forma alternada. Para informar al objeto que los datos están alternados se utiliza el atributo *SerieAlternada* —línea33— cargada mediante el método *poner_Alternada* —línea 15—. En la línea 17 está la sobrecarga del operador indexación. Este operador va a realizar todo el trabajo relacionado con el acceso a datos. Desde el punto de vista del usuario su uso es el habitual; va a utilizar el objeto de tipo *Serie* como de un *array* se tratase. El operador sobrecargado recibe como argumento el índice del dato a ser accedido, pero debe interpretar a que dato hace referencia este índice. Para ello lo primero que realiza es discriminar qué tipo de dato se está tratando, consultado el valor del atributo *tipoDato* —líneas 20, 22 y 24—. Luego determina si la serie contiene un único conjunto de datos o contiene dos conjuntos de datos alternados —operador ternario en las líneas 21, 23 y 25—. Si el dato es de 8 bits, convierte el puntero «void» *Datos* en puntero a datos de tipo «uint8_t» —8 bits—. De forma análoga sucede en los caso de 16 y 32 bits donde el puntero es convertido a tipo «uint16_t*» y «uint32_t*» respectivamente. Si la serie es alternada, al índice recibido lo multiplica por dos, mediante el desplazamiento de un bit a la izquierda; de esta forma se accede a la posición 0, 2, 4, 6,... del *array*. Si lo que se desea son las posiciones impares, en la carga de los datos, se debe especificar la dirección del elemento en la posición 1 del *array* en la llamada al método *ponDatos*. Para manejar esta nueva situación de forma sencilla en la carga de datos, se suministran otras tres macros —líneas 1, 3 y 5—. Al igual que las macros anteriores, se trata de tres versiones para diferentes tipos de datos. Su prototipo es *CARGA_ARRAY_ALTERNADO_XBITS*, donde «XBITS» puede ser «8BITS», «16BITS» o «32BITS» para tipos de datos de 8, 16 o 32 bits respectivamente. Estas macros tienen dos argumentos: el nombre del *array* donde están almacenados los datos, y si se accede a datos pares o

[20] Modo en el que dos o más ADCs convierten simultáneamente de forma sincronizada, entregando los datos a través de DMA en forma alternada.

impares. Si se desea acceder a datos pares, el segundo argumento ha de ser cero, sino, ha de ser 1. Un ejemplo de uso de lo explicado se muestra en el siguiente fragmento de código:

```
1  uint8_t fuenteDatos[4096];
2
3  Graph::Serie serie1;
4  Graph::Serie serie2;
5
6  serie1.ponTipoDato(Graph::Serie::PUNTERO_16BITS);
7  serie1.ponDatos(CARGA_ARRAY_ALTERNADO_16BITS(fuenteDatos,0));//se accede a datos pares
8  //equivale a: serie1.ponDatos(fuenteDatos, 1024);
9
10 serie2.ponTipoDato(Graph::Serie::PUNTERO_16BITS);
11 serie2.pomDatos(CARGA_ARRAY_ALTERNADO_16BITS(fuenteDatos,1));//se accede a datos impares
12 //equivale a: serie2.ponDatos(&fuenteDatos[2], 1024);
13 //o a:        serie2.ponDatos(&((uint16_t *)fuenteDatos[1]), 1024);
14
15
16 uint16_t elemento_par_1=serie1[0];
17 uint16_t elemento_par_2=serie1[1];
18
19 uint16_t elemento_impar_1=serie2[0];
20 uint16_t elemento_impar_2=serie2[1];
```

Código 4.61: Carga de datos alternados en el objeto de tipo «Serie»

En las líneas 3 y 4 se declaran los objetos *serie1* y *serie2* de tipo *Serie*. En las líneas 6 y 7 se cargan los datos pares del *array fuenteDatos* tratados como tipos de 16 bits, mientras en las 10 y 11 se cargan los impares —tratado como datos de tipo 16 bits—. Cabe destacar la sintaxis alternativa que aparece en los comentarios del código, especialmente los mostrados en las líneas 12 y 13, donde se aprecia la mayor complejidad sintáctica —cuando se evita el uso de las macros—. En las líneas 16 y 17 se muestra como se accede a los dos primeros datos pares y en las 19 y 20 a los dos primeros impares. Desde el punto de vista del programador se trata de dos grupos independientes de datos y su acceso se realiza mediante el operador de indexación, tal como si de dos *arrays* independientes se tratase.

4.4.1.11.2 Representación de datos.

Una vez tratado el acceso a los datos, viene el proceso de representarlos en pantalla. La acción a tomar es determinar cuál es la correspondencia entre los datos almacenados en memoria y los puntos en pantalla. La primera aproximación es hacer corresponder cada posición de los datos en la serie con cada posición horizontal de puntos en pantalla.

La captura se realiza sobre 2048 datos de tipo flotante de 32 bits, pero una vez realizada la Transformada Rápida de Fourier, debido a que estos son capturados desde una función real —la señal temporal—, el número de datos útiles es de 1024, ya que el espectro es simétrico —respecto del cero—, y al ser la longitud horizontal de pantalla de 480 puntos, permite repartir la representación del espectro en $2,\widehat{13}$ pantallas. Esto haría que se viese una primera pantalla con los primeros 480 puntos del espectro —desde el dato 0 al 479—, una segunda pantalla con los 480 siguientes —desde el dato 480 al 959— y una última con los 64 puntos restantes — desde el dato 960 al 1023 del espectro—. Con ello se tendría la representación del espectro repartida en tres pantallas sin continuidad entre cada una de ellas; lo que se pretende es poder desplazar los datos por pantalla sin que haya una correspondencia tan estricta entre datos y puntos de pantalla, manteniendo la relación uno a uno —un punto de pantalla por cada dato—. Para ello se establece que el inicio del registro de datos corresponda al inicio horizontal de pantalla, pero cada dato no tenga una posición fija en pantalla, sino que pueda adquirir cualquier posición, permitiendo el desplazamiento de la representación sobre ella. El desplazamiento mínimo es de un punto, con lo que en total se tienen 544 desplazamientos posibles —el total del registro, 1024, menos el ancho horizontal de pantalla 480—. Si se implementa la realización de este desplazamiento mediante un control gráfico,

como pueda ser un botón, se necesitarían 544 eventos «click» para recorrer todo el registro. Por ello el desplazamiento puede realizarse en paquetes de puntos, aunque esto será matizado más adelante:

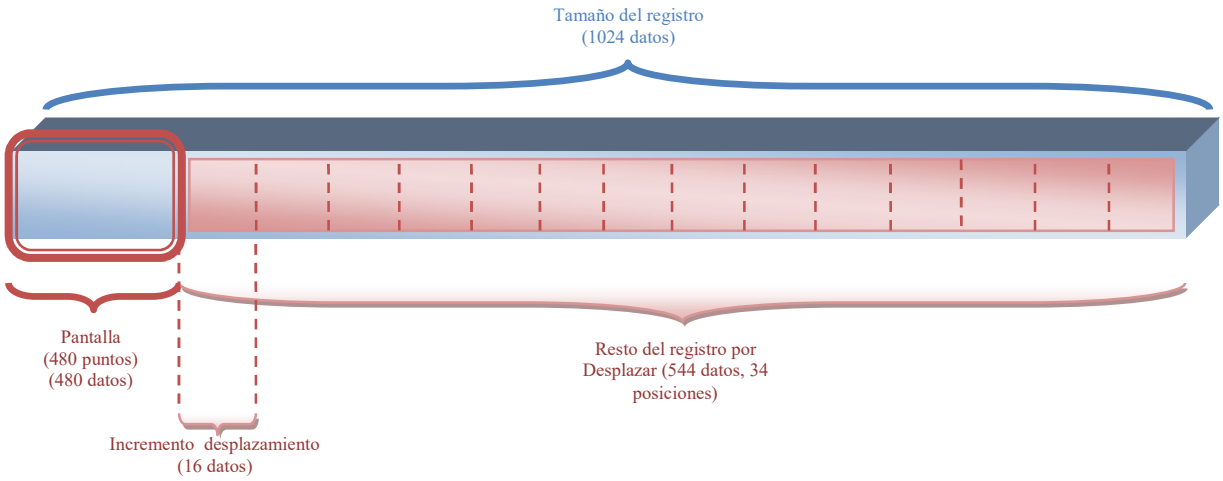


Ilustración 4.22: Desplazamiento de la pantalla sobre el registro referido al inicio del mismo

En esta figura están representados tanto el registro —color azul— como la pantalla —color rojo—. La pantalla muestra los 480 primeros datos del registro, lo que supone que el desplazamiento de la pantalla respecto del registro es cero. La zona sombreada en rojo muestra la zona del registro no visualizada —544 datos—, zona en la que se puede desplazar la pantalla, y como se comentaba anteriormente, si el desplazamiento se hace punto a punto, hay 544 desplazamientos. Así si desplazamos la representación de la pantalla al punto 100, los datos visualizados serán los correspondientes a las posiciones de la 100 a la 579. Para mayor versatilidad en el desplazamiento se puede realizar en grupos de 16 puntos quedando un total de 34 —544/16— desplazamientos adicionales, además de la posición cero. De forma esquemática, estas posiciones están representadas en la ilustración anterior como líneas discontinuas rojas sobre la parte del registro no visualizada —no hay que perder de vista que se trata de realmente de una representación esquemática—.

Sin embargo este no es el esquema finalmente implantado de representación y desplazamiento de la pantalla sobre el registro. Lo interesante es que el índice de referencia del registro, desde donde se tome el desplazamiento, sea el centro del mismo. De igual forma, el punto de referencia para la representación en pantalla no sea la zona horizontal izquierda sino el centro de pantalla. Es decir, el desplazamiento cero debe estar referido al centro de la pantalla respecto del centro del registro. La elección de este esquema es debida a que los osciloscopios digitales siguen esta misma filosofía, y aunque finalmente no se desarrolla una aplicación de tipo osciloscopio, la presentación gráfica de un espectro de frecuencias suele estar referida a una frecuencia que está en el centro de la pantalla —frecuencia central—, con lo que este modelo es el conveniente.

De esta forma, el índice de referencia del registro es el 512 —1024/2— quedando desde las posiciones 0 a la 511 como índices negativos —512 puntos— y de la 513 a la 1023 como positivos —511 puntos—. Sin embargo, ya que finalmente lo que se indexa es un *array*, se deben traducir estas posiciones «relativas» a posiciones de este *array* —índice del mismo—. En el siguiente esquema se muestra la relación entre la posición de la pantalla y el registro:

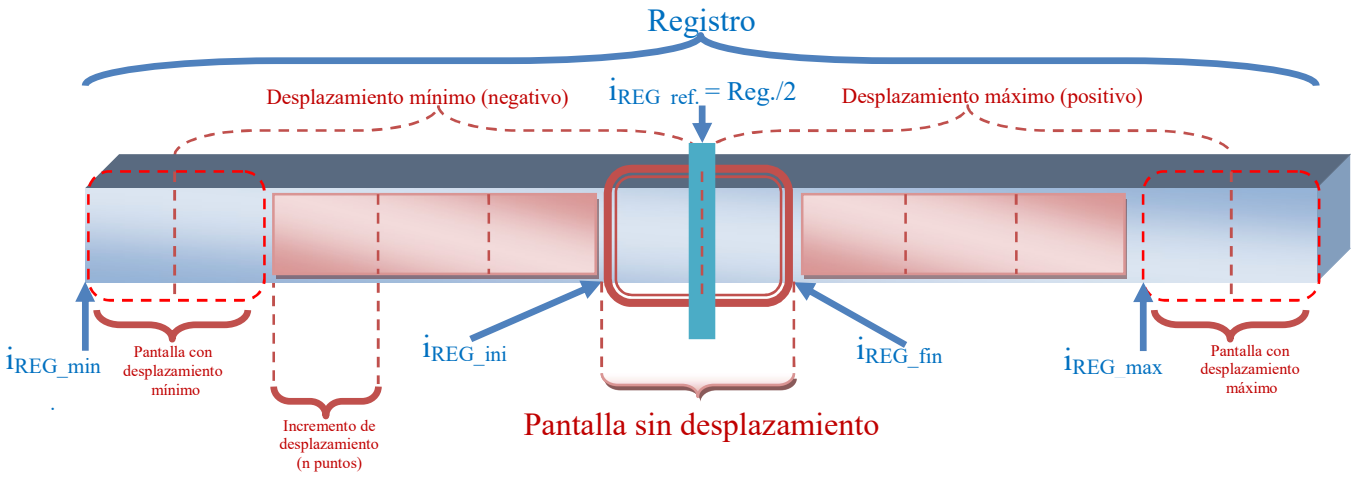


Ilustración 4.23: Desplazamiento del centro de pantalla sobre el registro referido a la mitad del mismo.

En esta figura está marcado el centro del registro —mediante una línea ancha azul en su centro y denotado como i_{REG_ref} (índice de referencia del registro) —, que corresponde con el dato a mostrar en el centro de pantalla cuando no hay añadido ningún desplazamiento. Numéricamente, su valor será el tamaño del registro partido por dos —se sigue manteniendo el esquema en el que cada dato del registro es un punto en pantalla—. A pesar de querer tener como referencia la parte central de la pantalla, se necesita saber cuál es el índice del registro que correspondería al punto más a la izquierda de la misma. Este punto está nombrado como i_{REG_ini} sobre la figura. Su valor será el del índice de referencia del registro —i_{REG_ref}— menos la mitad del ancho de pantalla. Otro índice del registro a conocer, aunque de menos importancia que los anteriores, es el índice del dato que se pinta en el último punto de pantalla —denotado como i_{REG_fin}—. En definitiva, los datos del registro a representar en pantalla, cuando esta no se ha desplazado del centro del registro, son los comprendidos entre los índices i_{REG_ini} e i_{REG_fin} del registro. Si se desea desplazar la pantalla por el registro, se necesita añadir a este rango de índices un valor adicional que indique este desplazamiento —posición horizontal—. Si se desplaza la pantalla hacia la zona derecha del registro, este valor es positivo, pero si se hace hacia la izquierda, es negativo. Conociendo esto, se pueden determinar la posición del índice inicial y final que aparece en pantalla para un desplazamiento dado:

$$i_{REG\_ref} = \frac{REG}{2} \tag{4.2}$$

$$i_{REG\_ini} = \frac{REG}{2} - \frac{PAN}{2} + DES = i_{REG\_ref} - \frac{PAN}{2} + DES \tag{4.3}$$

$$i_{REG\_fin} = i_{REG\_ini} + PAN = \left(\frac{REG}{2} - \frac{PAN}{2} + DES\right) + PAN = i_{REG\_ref} + \frac{PAN}{2} + DES \tag{4.4}$$

En las dos últimas expresiones se dan los valores de los índices inicial y final del registro que aparecen en pantalla en función del índice de referencia —valor constante— y del desplazamiento deseado «*DES*»—negativo si es hacia la izquierda y positivo si lo es hacia la derecha—. «*REG*» hace referencia al tamaño del registro —en número de datos—, mientras que «*PAN*» lo hace al de pantalla —en número de puntos horizontales de pantalla o del ancho del control utilizado como gráfica—. Este desplazamiento debe ser tal, que no se intenten representar más allá del índice mínimo del registro sobre pantalla —i_{REG_min}— ni del máximo —i_{REG_max}—. El índice mínimo es el cero, pero el máximo, será el máximo del registro, menos el ancho de pantalla. Teniendo esto en cuenta y las dos últimas expresiones anteriores, se pueden determinar los valores máximo y mínimo para el desplazamiento, ya que para el desplazamiento mínimo, el índice inicial se convierte en el índice mínimo de pantalla y para el desplazamiento máximo, el índice final es el índice máximo:

$$i_{REG\_min} = 0 \tag{4.5}$$

$$i_{REG\_max} = REG - PAN \tag{4.6}$$

$$i_{REG\_min} = i_{REG\_ref} - \frac{PAN}{2} + DES_{min} \rightarrow 0 = i_{REG\_ref} - \frac{PAN}{2} + DES_{min} \rightarrow \ DES_{min} = \frac{PAN}{2} - i_{REG\_ref} \tag{4.7}$$

$$\begin{gathered} i_{REG\_fin} = i_{REG\_ref} + \frac{PAN}{2} + DES_{max} \rightarrow REG - PAN = i_{REG\_ref} + \frac{PAN}{2} + DES_{max} \\ \rightarrow DES_{max} = REG - PAN - \frac{REG}{2} - \frac{PAN}{2} \rightarrow \ DES_{max} = i_{REG\_ref} - \frac{PAN}{2} \end{gathered} \tag{4.8}$$

Así, para el caso del registro de 1024 datos y el ancho horizontal de pantalla de 480 puntos, se tienen los siguientes límites en los valores de desplazamiento horizontal:

$$DES_{min} = \frac{PAN}{2} - \frac{REG}{2} = \frac{480}{2} - \frac{1024}{2} = -272 \tag{4.9}$$

$$DES_{max} = \frac{REG}{2} - \frac{PAN}{2} = \frac{1024}{2} + \frac{480}{2} = 272 \tag{4.10}$$

Como se aprecia, el desplazamiento máximo es positivo —desplazamiento hacia la derecha— mientras que el mínimo es negativo —desplazamiento hacia la izquierda—. Se reitera que estos valores son para el caso de haber correspondencia uno a uno entre datos del registro y puntos de pantalla con incremento de desplazamiento igual a uno, lo que implica que se puede desplazar la pantalla sobre el registro en 544 posiciones diferentes —272-(-272)=544—. Bajo estas condiciones se puede calcular, igualmente, los índices del registro que ocuparán las posiciones centrales en pantalla para cualquier desplazamiento y en concreto para los dos desplazamientos extremos:

$$i_{CENTRAL} = i_{REG\_ref} + DES \tag{4.11}$$

$$i_{CENTRAL\_min} = i_{REG\_ref} + DES\_\min = 512 + (-272) = 240 \tag{4.12}$$

$$i_{CENTRAL\_max} = i_{REG\_ref} + DES\_\max = 512 + (+272) = 784 \tag{4.13}$$

Con el índice central determinado para un valor concreto de desplazamiento, finalmente se calcularía el índice inicial del registro sobre pantalla —i_{REG_ini}— que permite dibujar la gráfica pasando los 480 datos consecutivos del registro —desde este índice en adelante— a la pantalla.

Sin embargo, la correspondencia de datos del registro y puntos en pantalla no va a ser siempre de uno a uno ya que se pretende poder expandir y contraer el gráfico horizontalmente. Cuando se contraiga el gráfico, varios datos del registro ocuparán horizontalmente los mismos puntos en pantalla —no así verticalmente—, mientras que si se expande, los datos contiguos del registro estarán separados por varios puntos en pantalla. A este nuevo

parámetro se le puede denominar «escala horizontal del gráfico», de tal forma, que si este tiene valor uno, significará que hay correspondencia uno a uno entre los datos y los puntos; si es mayor a uno, significará que hay varios datos representados sobre un mismo punto horizontalmente y si es menor a uno significará que la inversa de su valor representa la separación de puntos en pantalla entre datos contiguos del registro. Por ejemplo, en este proyecto los posibles valores para esta escala horizontal son: 2, 1, 1/2, 1/3, 1/4, 1/5 y 1/6, significando, respectivamente, dos datos sobre el mismo punto horizontal de pantalla; cada dato está representado en la horizontal de cada punto de pantalla; cada dato del registro está separado del siguiente por dos puntos en pantalla, cada dato del registro está separado del siguiente por tres puntos en pantalla, y así sucesivamente. Cuando el valor de esta escala sea mayor a uno, el incremento de desplazamiento será igual al número de puntos sobre la misma horizontal, es decir, que los puntos que se representan sobre la misma horizontal se desplazan a la vez. El valor de este incremento coincide con el valor de la escala. En el caso de este proyecto el valor de escala horizontal máximo es igual a 2. En este caso el incremento de desplazamiento será de 2 —los dos puntos sobre la misma horizontal se desplazan a la vez—. Esto hace que se indexen los datos de índice par del registro y en la función *draw* —la encargada de realizar la representación del control— se pinten cada uno de ellos en pantalla y sobre su misma horizontal, los impares —si la escala hubiese sido de tres, por ejemplo, se indexarían los índices múltiplos de tres y en pantalla la función *draw* los representaría sobre puntos horizontales sucesivos y se encargaría de representar los dos siguientes del registro sobre la misma horizontal. En este sentido, desde el punto de vista del registro, la pantalla queda ampliada en un factor igual a la escala horizontal, ya que por ejemplo, si la escala horizontal es igual a dos, y la pantalla tiene un ancho de 480 puntos, en ella se van a representar 960 datos del registro. Cuando su valor sea uno o menor, el incremento de desplazamiento será de uno —un punto de pantalla de desplazamiento—. En el caso de la escala igual a uno, cada desplazamiento de un punto en pantalla corresponderá al desplazamiento de cada dato del registro indexado. Para el caso de valores de la escala horizontal menor a uno, un desplazamiento de un punto en pantalla significará que se necesitan tantos desplazamientos como la inversa del valor de la escala horizontal para que un dato del registro adquiera la posición del valor inmediatamente anterior o posterior del registro —según el sentido del desplazamiento—. Por ejemplo, con una escala horizontal igual a 1/3 —cada valor del registro está separado de los adyacentes en tres puntos de pantalla—, para que un valor concreto representado adquiera la posición del que lo está posteriormente a él, se necesitan tres desplazamientos —los tres puntos que los separan—. Otra consecuencia del uso de la escala horizontal, cuando es menor a uno, es que desde el punto de vista de la pantalla el registro queda ampliado con un factor de valor igual a la inversa de esta escala. Si la escala es de 1/4 —cada valor del registro está separado de los adyacentes en cuatro puntos en la pantalla—, equivale a decir que el tamaño del registro ha crecido en un factor igual a cuatro desde el punto de vista de la pantalla —donde antes representaba la pantalla «n» puntos con una escala horizontal igual a uno, con una escala igual a 1/4 se necesitan cuatro pantalla para representar esos «n» puntos—. Teniendo en cuenta la escala horizontal, las expresiones de desplazamiento mínimo y máximo quedan finalmente como:

$$DES_{max} = \frac{REG \times \frac{1}{escala\_horizontal}}{2} - \frac{PAN}{2} \tag{4.14}$$

$$DES_{min} = \frac{PAN}{2} - \frac{REG \times \frac{1}{escala\_horizontal}}{2} = -DES_{max} \tag{4.15}$$

Estas dos expresiones generalizan los desplazamientos máximos y mínimos para escalas horizontales menores a uno —la escala horizontal de valor uno también cumple estas expresiones—. Se recuerda que en este caso, donde la correspondencia entre datos y puntos es de uno a varios —escala menor a uno—, los desplazamientos hacen referencia a puntos de pantalla y se necesita un incremento en el desplazamiento igual a la inversa de la escala para indexar o hacer referencia a un nuevo dato del registro en pantalla. Teniendo en cuenta que se hace referencia a un dato cada inversa de la escala horizontal, se puede calcular el índice central para un desplazamiento dado, e igualmente los índices centrales máximo y mínimo:

$$i_{CENTRAL} = \left\lfloor i_{REG\_ref} + DES \times escala\_horizontal \right\rfloor \quad (4.16)$$

$$i_{CENTRAL\_min} = \left\lfloor i_{REG\_ref} + DES_{min} \times escala\_horizontal \right\rfloor \quad (4.17)$$

$$i_{CENTRAL\_max} = \left\lfloor i_{REG\_ref} + DES_{max} \times escala\_horizontal \right\rfloor \quad (4.18)$$

El índice central de la pantalla será el índice de referencia del registro —el que está en medio del registro— más el desplazamiento por la escala horizontal —que es menor a uno—, tomando solo la parte entera de esta operación. De esta forma, el índice central solo se incrementará al haberse producido tantos desplazamientos como indique el valor de la escala horizontal. Así, por ejemplo, si la escala horizontal es ¼, se necesitará cuatro incrementos de desplazamiento para que el índice central se incremente en una unidad apuntando al siguiente dato.

Cuando la escala horizontal es mayor a uno, varios datos consecutivos del registro son representados en el mismo punto horizontal de pantalla —como máximo 2, en este proyecto—, lo que equivale a decir que el ancho de pantalla crece en un factor igual a la escala horizontal de pantalla:

$$DES_{max} = \frac{REG}{2} - \frac{PAN \times escala\_horizontal}{2} \quad (4.19)$$

$$DES_{min} = \frac{PAN \times escala\_horizontal}{2} - \frac{REG}{2} = -DES_{max} \quad (4.20)$$

El índice central se determina como en el caso de la escala horizontal igual a uno, ya que el gobierno del indexado en la cantidad igual a la escala, se produce al incrementar/decrementar el desplazamiento —ver lo explicado sobre los métodos i*ncrementaPosH* y *decrementaPosH*, más abajo—. Se recuerda que el incremento de puntos en el desplazamiento para esta escala mayor a uno, es su valor. Con estas expresiones se calculan los desplazamientos mínimo y máximo e índices centrales mínimo y máximo de las distintas escalas horizontales, resumidas en la siguiente tabla:

Escala Horizontal (datos por punto)	Des_{min} (puntos de pantalla)	Des_{max} (puntos de pantalla)	$i\_central_{min}$ (posición en el registro)	$i\_central_{max}$ (posición en el registro)
2	-32	32	480	544
1	-272	272	240	784
1/2	-784	784	120	904
1/3	-1296	1296	80	944
1/4	-1808	1808	60	964
1/5	-2320	2320	48	976
1/6	-2832	2832	40	984

Tabla 18: Desplazamientos e índices centrales respecto a la escala horizontal del gráfico

Para manejar los desplazamientos horizontales, la clase *Serie* dispone de los siguientes atributos y métodos:

```
1  class Graph : public Widget
2  {
3      public:
4          . . .
5          class Serie
6          {
7              . . .
8              public:
9                  //métodos HORIZONTAL
10                 void ponPosicionHor(int16_t posicion);
11                 int16_t obtenPosicionHor(void) {return DesplazamientoHor;}
12                 int16_t obtenDesplazamientoHor_min(void) {return DesplazamientoHor_min;}
13                 int16_t obtenDesplazamientoHor_max(void) {return DesplazamientoHor_max;}
14                 void ponEscalaHor(float escala);
15                 float obtenEscalaHor(void) {return EscalaHor;}
16                 void determinaLimitesPosicionHor(void);
17                 void incrementaPosH(float inc=1);
18                 void decrementaPosH(float dec=1);
19             . . .
20             protected:
21                 //atributos HORIZONTAL
22                 uint16_t i_ref;
23                 int16_t DesplazamientoHor;
24                 int16_t DesplazamientoHor_max;
25                 int16_t DesplazamientoHor_min;
26                 float EscalaHor;
27             . . .
28         }
29 };
```

Código 4.62: Control de la parte horizontal en la clase «Serie»

El atributo *i_ref* —línea 22— contendrá el índice del dato del registro que se localiza en su mitad; que para el caso concreto que se trata, es la constante 512. Este valor será cargado en el proceso de asociación entre objeto *Serie* y objeto *Graph* —la serie es añadida al gráfico—. La escala actual en uso es almacenada en el atributo *EscalaHor* —línea 26— cuyos posibles valores, ya comentados, son: 2, 1, 1/2, 1/3, 1/4, 1/5 y 1/6. Estos valores serán asignados al atributo mediante el método *ponEscalaHor* y accedidos mediante *obtenEscalaHor* —líneas 14 y 15, respectivamente—. El método *ponEscalaHor*, además, llama al método *deteminaLimitesPosicionHor* —línea 16— que establece los límites máximo y mínimo de los desplazamientos para la escala horizontal en uso, almacenado los resultados en los atributos *DesplazamientoHor_max* y *DesplazamientoHor_min* —líneas 24 y 25 respectivamente—. El desplazamiento actual en uso está almacenado en el atributo *DesplazamientoHor* y es cargado y accedido mediante los métodos respectivos *ponPosicionHor* y *obtenPosicionHor* —líneas 10 y 11—. Finalmente se facilitan dos métodos, *incrementaPosH* y *decrementaPosH* —líneas 17 y 18— que permiten incrementar y decrementar la posición teniendo en cuenta la escala en uso. Para el caso de un valor de la escala mayor o igual a uno, el incremento/decremento es el valor de la escala. Por ejemplo, si la escala es igual a dos, el incremento/decremento será de dos —dos datos en cada punto de pantalla—. Esto hace que se indexe datos en el registro cada dos posiciones del mismo —el dato saltado se encargará el método *draw* de dibujarlo en la misma posición horizontal que el dato realmente indexado—. En cambio, si la escala es menor a uno, el incremento/decremento es de uno —un dato en cada punto de pantalla aunque la separación entre dato y dato sea el valor de la inversa de la escala—. Estos dos últimos métodos son los realmente utilizados en este proyecto para variar el desplazamiento. Estas dos funciones hacen uso de *ponPosicionHor* y *obtenPosicionHor*. El método *obtenPosicionHor* simplemente devuelve el valor del atributo *DesplazamientoHor*, sin embargo *ponPosicionHor* realiza una comprobación previa a la asignación del valor pasado a su argumento al atributo *DesplazamientoHor*. Esta comprobación consiste en verificar si el nuevo desplazamiento a aplicar está dentro de los límites del desplazamiento permitido —*DesplazamientoHor_max* y *DesplazamientoHor_min*—. Si es así procede a la asignación del nuevo desplazamiento, sino, pone como desplazamiento el límite del desplazamiento rebasado —*DesplazamientoHor_max* o *DesplazamientoHor_min*—. Esta forma de actuar permite, por un lado, no rebasar estos límites y por otro, poder representar el primer y último dato del registro en el caso de utilizar un valor para la escala horizontal que no divida de forma exacta al registro —no es el caso de este proyecto ya que el valor máximo de escala horizontal es de de dos, que divide de forma exacta la longitud del registro—.

Una vez determinado como se trata la escala y posición horizontal y la correspondencia entre datos del registro y puntos a mostrar en pantalla, se han de dibujar estos puntos. De ello se encarga el método *draw* del que disponen todos los componentes gráficos de la librería *TouchGFX* así como los creados a partir de la clase *Widget*, como es el caso. Esté método ya ha sido descrito en la sección 1.13.3, pero cabe recordar que es un método llamado por la librería y para que su ejecución esté repartida entre los diferentes componentes gráficos de la pantalla y la representación de todos se haga «a la vez», *TouchGFX* manda dibujar cada componente dividido en regiones de tal forma que para dibujar la totalidad de un componente se necesitarán varias órdenes de dibujado —para pintar cada una de las regiones gráficas en que la librería subdivida al componente—. Por tanto, cuando *TouchGFX* manda dibujar una región del componente, ha de especificar la dimensión y posición de la misma como argumento al método que el componente utiliza para redibujarse; este método es *draw*. La región está descrita mediante un objeto de tipo *Rect* pasado como argumento al método anterior, tal y como fue explicado en la sección 1.13.3. Hay que recordar, también, que el objeto de la clase *Serie* contendrá información para dibujar la serie gráfica que representa, pero no es el encargado del dibujado de la misma. El responsable de ello será el objeto de la clase *Graph*, clase que deriva de *Widget* y por tanto dispone del método *draw*. El objeto de la clase *Graph* accederá a información del objeto/s *Serie* contenido/s para dibujarlo/s. Tal es el acceso del objeto *Graph* a la información del objeto *Serie*, que la clase del primero está definida como «amiga» de la clase del segundo para poder acceder a sus partes protegidas:

```
1  class Graph : public Widget
2  {
3      public:
4          . . .
5          class Serie
6          {
7              . . .
8              public:
9              . . .
10             protected:
11                  //atributos OBJETO EXTERNO
12                 Graph* grafico;
13                 friend class Graph;//posibilita que el objeto externo de tipo Graph pueda
14                                    // acceder a las partes protegidas del objeto de la clase Serie
15             . . .
16         }
17     . . .
18 };
```

Código 4.63: Declaración de la clase «Graph» como amiga de la clase «Serie»

Ya que la representación de los datos contenidos en el registro —registro referenciado por el objeto de la clase *Serie*— va a depender si la escala horizontal es mayor o menor a uno, el tratamiento en el método *draw* tendrá en cuenta esta distinción. El esqueleto de este método es el siguiente:

```
1  void Graph::draw(const Rect& invalArea) const
2  {
3     //se bloquea el buffer secundario
4      uint16_t* frameBuffer = (uint16_t*) HAL::getInstance()->lockFrameBuffer();
5
6      int y;
7      static int y_ant;
8      int i;
9      float DatosPorPunto;
10     float PuntosPorDato;
11     Serie *serie;
12
13
14     Rect absoluteRect = getAbsoluteRect();
15
16     for (int n=0; n < nSeries; n++)
17     {
18         serie=Series[n];
19         DatosPorPunto=serie->EscalaHor;
20         PuntosPorDato=1/serie->EscalaHor;
21
22         //si hay que dibujar uno o varios datos en el mismo punto horizontal
23         if ((int)DatosPorPunto>=1)          . . .
24          . . .
25
26         // si hay que dibujar cada dato separado por varios puntos de pantalla
27         else
28          . . .                              . . .
29
30     }
31     //se desbloquea el buffer secundario
32     HAL::getInstance()->unlockFrameBuffer();
33 }
```

Código 4.64: Esqueleto del método «draw» de la clase «Graph»

Lo primero que recibe el método *draw* es la referencia del objeto de tipo *Rect*, llamado *invalArea*, como argumento pasado directamente de la capa *CORE* de la librería y que representa la región de la gráfica a ser redibujada —línea 1—.

Como se dibuja directamente sobre el buffer secundario, se ha de asegurar que nadie más esté dibujado sobre él, concretamente que no lo esté haciendo el DMA, por ello se llama a la función de la capa HAL *lockFrameBuffer*, que no retorna hasta que dicho buffer está disponible y lo bloquea hasta que el método *draw* termine su trabajo —línea 4—. También devuelve la dirección actual del buffer secundario en uso —ya que este buffer no es siempre el mismo debido a que se intercambia continuamente con el buffer que el controlador de pantalla LTDC[21] está usando para refrescarla (técnica de doble buffer) —. Entre las líneas 6 a 11 se declaran varias variables necesarias para el proceso de redibujado. La variable *y* —línea 6— almacenará la posición vertical de los datos del registro en cada instante mientras que la variable *y_ant* —línea 7— contendrá los valores verticales del dato anterior al tratado actualmente —*y*—, cuya utilidad es la de poder trazar vectores entre el dato tratado actualmente y el tratado con anterioridad. Estas dos variables son de tipo entero ya que las posiciones en pantalla también lo son, aunque la tarea encargada de la adquisición y manipulación de datos los trata como flotantes internamente. Las variables *DatosPorPunto* y *PuntosPorDato* —declaradas en las respectivas líneas 9 y 10— son distintas representaciones de la escala horizontal para mejorar la manipulación de los datos y la legibilidad del código. Así la variable *DatosPorPunto* adquiere directamente el valor de la escala horizontal —línea 19— ya que el valor de la escala horizontal significa cuantos puntos contiguos del registro se dibujan sobre el mismo punto horizontal. Esta representación es conveniente para el caso en que la escala horizontal es mayor o igual a uno —línea 23—, sin embargo cuando la escala horizontal es menor a uno —línea 27— , datos contiguos del registro están separados en pantalla por varios puntos, en concreto tantos puntos como la inversa de la escala horizontal. Este es el valor con que se carga la variable *PuntosPorDato* en la línea 20. En cualquier caso, ya que se ha de dibujar en el buffer secundario, se necesita conocer las coordenadas absolutas del componente al que dibujar una región. Ello se consigue mediante la llamada a la función *getAbsoluteRect* declarada en la clase

[21] Periférico del micro STM32F4 encargado de servir como controlador de display.

Widget, asignado las coordenadas absolutas del objeto *Graph* a tratar, a la variable *absoluteRect*, de tipo *Rect*. Para dibujar la información contenida en las distintas series añadidas en el objeto de tipo *Graph*, en su método *draw* se utiliza una sentencia de control «*for*», donde se recorre cada una de ellas, asignado sus respectivas direcciones al puntero «serie», definido en la línea 11. Así, las distintas formas de representar los datos —dependiendo de la escala horizontal— tendrán como único interfaz, para interaccionar con las series, este puntero. Finalmente, cuando el método *draw* ha terminado de redibujar la región solicitada, debe liberal el buffer secundario para que otros componentes puedan usarlo. Esto se hace con la llamada al método *unlockFrameBuffer* de la capa HAL en la línea 32.

En el caso de tener la escala horizontal igual o mayor a uno —línea 23—, se ha de determinar cuál es el índice del dato que se ha de representar para cada punto del gráfico, teniendo en cuenta el valor concreto de la escala, el desplazamiento utilizado, así como la coordenada horizontal del punto de pantalla a pintar. La parte del método *draw*, encargado de ello, es la que se muestra a continuación:

```
1  . . .
2  if ((int)DatosPorPunto>=1)
3   {
4    for (int x = invalArea.x; (x < (invalArea.x + invalArea.width) ) ; ++x )
5     {
6      for (int j=0;j<(int)DatosPorPunto;j++)
7        {
8          i=serie->i_ref-((serie->AnchoGrafico*serie->EscalaHor)/2)+\
9                                          serie->DesplazamientoHor+x*DatosPorPunto;
10         i=i+j;
11
12         if (i<0)continue;
13         if (i<serie->obtennDatos())
14          {
15            y=((*serie)[i]-serie->Ver_ref)*serie->EscalaVert+serie->DesplazamientoVer;
16            y=rect.height-y
17            if (serie->Vectores==false)
18             {
19               ponPunto(x, y, n, absoluteRect, invalArea, frameBuffer);
20             }
21            else
22             {
23               if (i==0)y_ant=y;
24               if (x>0)
25                {
26                  linea(x-1, y_ant, x, y, n, absoluteRect, invalArea, frameBuffer);
27                }
28                y_ant=y;
29             }
30          }
31         else break;
32       }
33     }
34   }
35 . . .
```

Código 4.65: Método «draw» de la clase «Graph» para escala horizontal mayor o igual a uno

Se recorre horizontalmente la región a redibujar —área invalidada— mediante el bucle *for* que aparece en la línea 4. El bucle *for* de la línea 6 junto con la sentencia de la línea 10, indexan tantos valores consecutivos del registro como el valor que la escala horizontal representa —*DatosPorPunto*—, para que sean pintados en la misma coordenada horizontal —variable «x» del bucle *for* de la línea 4—. Entonces, el índice de cada dato de este grupo que se representa bajo la misma coordenada horizontal, es determinado por un índice inicial del grupo —variable «i» actualizada en la línea 8— y un índice dentro de este grupo —variable «*j*» del bucle *for* de la línea 6—. La variable «*j*» es añadida a la «*i*», con lo que finalmente esta última es la que indexa el registro. Bajo la condición de este fragmento de código, solo será tratada la representación de los datos para los valores de escala horizontal 1 y 2, con lo que, para el primer valor no habrá grupo de datos consecutivos del registro dibujados bajo la misma coordenada horizontal, y para el segundo, este grupo será únicamente de dos datos. Aún así, el código se crea para tener versatilidad y poder manejar valores de escala horizontal mayores —*DatosPorPunto*—.

Para determinar el índice inicial del grupo, se toma como referencia el «índice de referencia» del registro —se recuerda que está localizado en la mitad del registro, y para el caso de este proyecto, es 512—, al que se le resta el valor de la mitad de la pantalla ampliada por la escala horizontal —tal y como ya se ha comentado— para que el desplazamiento esté referido a la parte izquierda y no a la central de pantalla, y finalmente se le añade el desplazamiento total a aplicar:

$$i_{REG\_ini} = \frac{REG}{2} - \frac{PAN \times escala\_horizontal}{2} + DES + x \cdot escala\_horizontal \tag{4.21}$$

Este desplazamiento total será la suma del desplazamiento aplicado —*DES*—, al utilizar los métodos *incrementaPosH* y *decrementaPosH*, que establecen el desplazamiento deseado desde el centro del registro, más el desplazamiento indicado por la distancia de la posición horizontal del punto de la región a pintar a la zona izquierda de la pantalla —$x \cdot escala\_horizontal$—. La variable *DES* ya está expresada en incrementos de la escala horizontal. Sin embargo la ordenada de pantalla —*x*—, ha de multiplicarse por la escala horizontal para que esté expresada en estos incrementos. Hay que tener en cuenta que la expresión anterior es para indexar el registro y no para determinar donde se pinta en pantalla .Esto está determinado por la variable x de la zona invalidada, pasada a los métodos *ponPunto* y *linea*, que son los que realmente pintan la información en pantalla. Como se ha comentado ya, el desplazamiento deseado ha sido corregido para estar referido a la parte izquierda de la pantalla y está especificado en paquetes de datos de tamaño igual al valor de la escala horizontal, de forma que cada incremento o decremento de desplazamiento indexa el registro en múltiplos de la escala horizontal. Como el valor máximo utilizado para la escala horizontal es dos, los datos que realmente se indexen serán los pares —los datos iniciales del grupo (que es de dos datos para la máxima escala) —. Para que el desplazamiento indicado por el punto actual a dibujar este indexando estos grupos de datos, se ha de multiplicar por la escala horizontal. La siguiente figura aclara lo explicado:

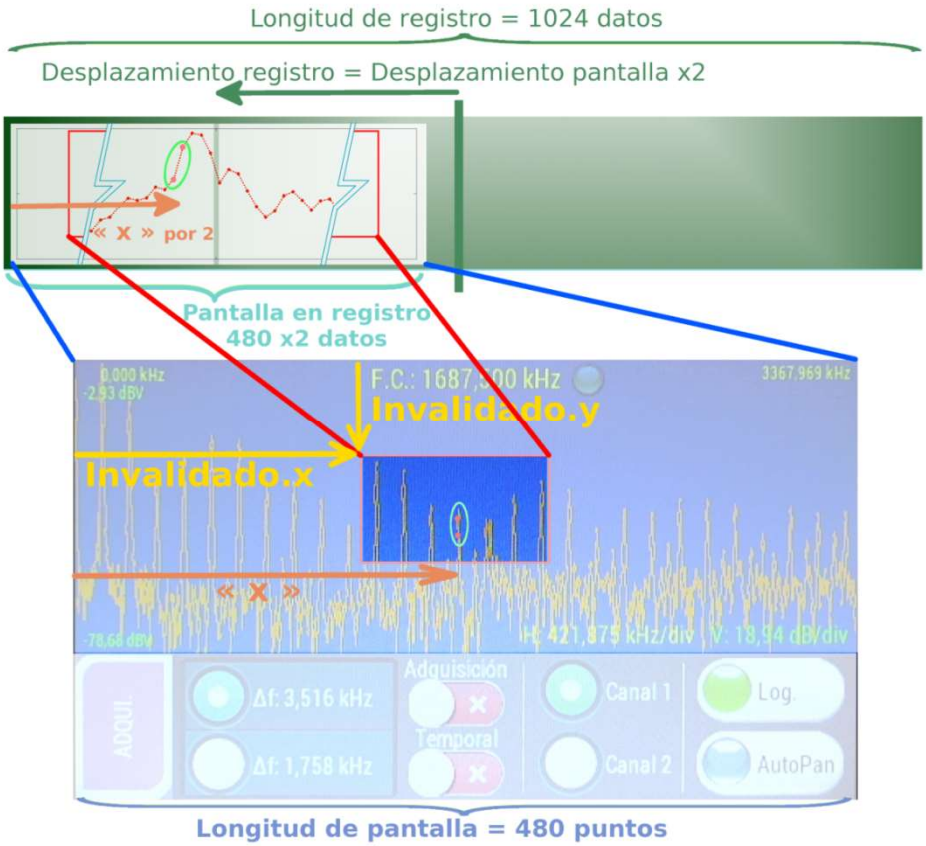


Ilustración 4.24: Relación entre registro y representación gráfica para escala horizontal igual a 2

Una vez determinado el índice del dato a pintar en la posición indicada por la variable «*x*» del bucle *for* de la línea 4, se procede a extraer ese valor del registro —línea 15—. A este valor se le aplica un factor —*EscalaVer*—, para variar la amplitud a la deseada y se le añade el desplazamiento vertical deseado —*DesplazamientoVer*—, ambos son atributos de la clase *Serie*. En cualquier caso, en pantalla, el eje «y» crece hacia abajo, lo que provocaría que la gráfica se viera invertida verticalmente. Para corregir esto se utiliza la sentencia de la línea 16, donde a la altura del gráfico se le resta el valor del dato indexado, de esta forma la gráfica se representa verticalmente «de abajo hacia arriba». Los datos procedentes del registro se pueden pintar como puntos aislados —línea 19— o como vectores que unan estos puntos —línea 26—. Para pintar solo puntos, se utiliza el método de la clase *Graph PonPunto*, donde sus argumentos iniciales son la posición del punto a dibujar «*x*» —referido en coordenadas relativas al componente gráfico—, el valor vertical del punto —«*y*»— y el número de orden de la serie a la que pertenece este punto —«*n*»—. Para dibujar un vector entre dos puntos se utiliza el método «línea» que además de los argumentos del método *PonPunto* necesita las coordenadas del punto anterior al tratado actualmente —«*x-1*» y «*y_ant*»—. Ambas funciones necesitan conocer las coordenadas absolutas del control —*absoluteRect*—, las coordenadas relativas del área a redibujar —*invalArea*— y la dirección del buffer gráfico secundario donde pintar —*frameBuffer*—. Todos estos datos han sido determinados dentro del método *draw* con anterioridad.

En el caso de ser la escala horizontal menor a uno, se necesita realizar cálculos de interpolación en el inicio y fin de la región a pintar, ya que, para la representación de vectores se necesita un punto anterior, y lo habitual es que el primer punto en la región no esté localizado en su parte más izquierda —por lo que se necesita interpolar el dato en la izquierda de la región, teniendo en cuenta el primer dato a pintar y el dato anterior del registro—. Cuando se pinta el último punto en la región sucede algo similar; no suele estar en la parte más a la derecha de la pantalla y se necesita interpolar este punto —teniendo en cuenta este último punto de la región y el siguiente del registro—.

El esquema de representación para una escala horizontal menor a uno es el siguiente:

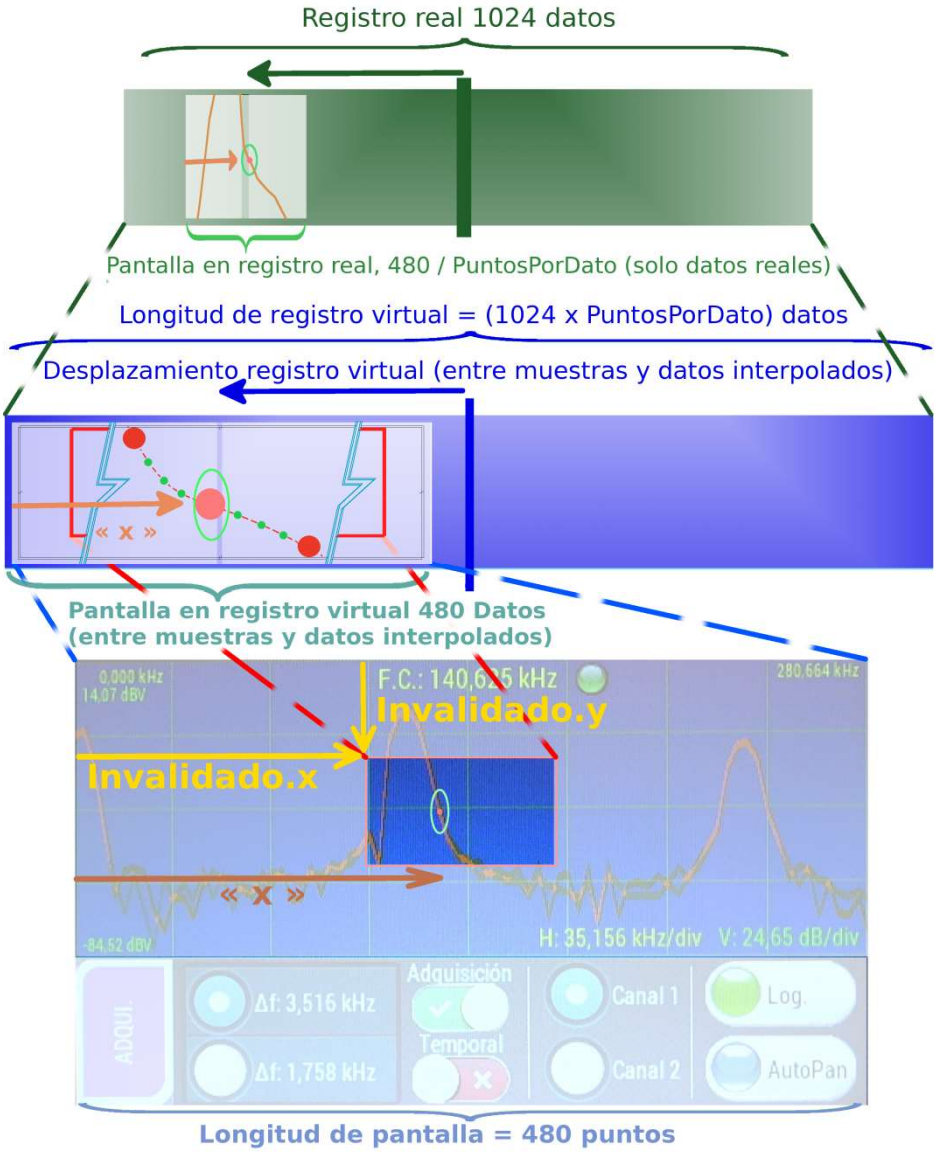


Ilustración 4.25: Relación entre registro y representación gráfica para escala horizontal menor a uno

En este caso, el registro, desde el punto de vista de la pantalla, se ha ampliado en un factor igual a la inversa de la escala horizontal —variable llamada *PuntosPorDato*—. Es equivalente a decir que la pantalla se ha reducido en este mismo factor respecto del registro, pero en este caso habría que tratar con posiciones fraccionarias en el registro, algo no conveniente. En la zona inferior de la ilustración anterior está dibujada la pantalla resaltando el área invalidada a ser pintado, y dentro de ella el punto actual a ser tratado. Están marcados, mediante flechas, las coordenadas del área invalidada —de forma relativa al control gráfico— y la coordenada horizontal del punto a ser dibujado. En la zona media de la ilustración se muestra el registro ampliado en el factor *PuntosPorDatos*. A este registro ampliado se le ha denominado «registro virtual», ya que es un registro con más datos de los que realmente tiene. Estos datos están constituidos por los datos reales más los datos interpolados entre las muestras. El «registro virtual» no existe físicamente en memoria, pero es un concepto útil para comprender el proceso de representación. Los datos reales son extraídos del registro real —zona superior de la ilustración— mientras que los datos interpolados son determinados en el momento de pintarlos. Aquí cabe distinguir entre los datos interpolados para unir dos muestras consecutivas —que se encarga de ello el método *línea*— y una interpolación previa que se realiza para determinar los puntos que corresponden a las coordenadas horizontales

extremas del área invalidada. Todo esto será aclarado, a continuación, cuando se muestre la parte del código que lo gestiona. En la parte de la ilustración del «registro virtual» se presenta el punto a ser tratado con color naranja —se ha representado para que coincida con una muestra del registro real—; las muestras anterior y posterior con color rojo y los espacios donde van los datos interpolados en color verde. La pantalla tiene de ancho 480 puntos, donde se mostrarán tanto muestras reales como interpoladas, pero en el registro real, el ancho de pantalla se ve reducido en un factor de *PuntosPorDato* que corresponde al número de puntos reales representados en pantalla. Desde el punto de vista del registro real la pantalla ha sido reducida, pero se asegura que se indexen posiciones enteras del mismo. Mientras que en el registro virtual, para una pantalla dada, se «indexan» un número de muestras reales y «huecos» correspondientes a las muestras interpoladas, procediendo al desplazamiento por pantalla de todas ellas, en el registro real solo se indexan las muestras reales, procediendo a realizar un desplazamiento de pantalla cuando se indexa una nueva muestra real. La parte del código del método *draw* que se encarga de hacer esto es el que se muestra a continuación:

```
1  . . .
2  else
3   {
4    static int x_ant=0;
5    int i_ant;
6    int i_pos;
7    int resto;
8    float m;
9
10   for (int x = invalArea.x; (x < (invalArea.x + invalArea.width) ); x++)
11    {
12     i=serie->i_ref*PuntosPorDato-((serie->AnchoGrafico)/2)+ serie->DesplazamientoHor+x;
13     //if((x==invalArea.x) || (invalArea.x + invalArea.width -1))
14     if ((x==0) || (x==this->getWidth()-1))
15      {
16       resto=i%(int)PuntosPorDato;
17       i_ant=i-resto;
18       i_pos=i_ant+PuntosPorDato;
19       m=((int)(*serie)[i_pos/PuntosPorDato]-\
20          (int)(*serie)[i_ant/PuntosPorDato])/PuntosPorDato;
21       y=m*resto+(*serie)[i_ant/PuntosPorDato];
22      }
23     else if ((i%(int)PuntosPorDato)==0)
24       y=(*serie)[i/PuntosPorDato];
25     else
26       continue;
27
28     y=(y-serie->Ver_ref)*serie->EscalaVert+serie->DesplazamientoVer;
29     y=rect.height-y;
30     if (serie->Vectores==false)
31      ponPunto(x, y, n, absoluteRect, invalArea, frameBuffer);
32     //else if(x>(invalArea.x))
33     else if (x>0)
34      linea(x_ant, y_ant, x, y, n, absoluteRect, invalArea, frameBuffer);
35
36     y_ant=y;
37     x_ant=x;
38    }
39   }
40  . . .
```

Código 4.66: Método «draw» de la clase «Graph» para escala horizontal menor a uno

Como en el caso del código que trataba la situación de una escala horizontal mayor o igual a uno, en este, donde se trata la situación de la escala horizontal menor a uno, lo primero que se hace es recorrer horizontalmente el área invalidada —bucle *for* de la línea 10—. Para cada valor que toma la variable x —y en consecuencia, para cada valor horizontal del área invalidada— se calcula el índice que correspondería al dato a indexar en el «registro virtual» —variable *i* de la línea 12—. Este está referido al índice de referencia del «registro virtual», que es el del registro real ampliado por el valor de la variable *PuntoPorDato*, por lo que es uno de los sumandos que componen el valor de este índice para que los datos estén contados desde el inicio de este registro y no desde la mitad del mismo. Cualquier desplazamiento se realiza en el «registro virtual» teniendo en cuenta el valor del desplazamiento horizontal deseado —*DesplazamientoHor*— y la coordenada horizontal del área invalidada a tratar en cada momento —*x*—. Para que este índice esté referido al inicio de la pantalla y no a

su parte media, al índice se le resta la mitad del ancho de pantalla. Hay que tener en cuenta que «*escala_horizontal*» es menor que uno y su inversa —denotada en el código como «*DatosPorPunto*»— es un valor entero:

$$i_{REG\_VIR\_ini} = \frac{REG}{2 \times escala\_horizontal} - \frac{PAN}{2} + DES + x \tag{4.22}$$

Pero este índice es el del «registro virtual», para determinar qué es lo que se indexa en el registro real se utilizan las sentencias entre las líneas 23 a 26. Estas sentencias pertenecen a un bloque condicional, siendo la condición principal la tratada en las líneas 13 y 14 que será comentada más adelante. En la línea 23 se comprueba si el índice del «registro virtual» corresponde con una posición del registro real. Esto se consigue verificando si el índice del «registro virtual» es múltiplo de la variable *PuntosPorDato* que representa los puntos en pantalla que separan dos muestras reales. Si es así, se determina el índice del registro real como el índice del virtual partido por *PuntoPorDato* en la línea 24—ya que es la relación de tamaños entre los dos registros y se ha verificado en el «registro virtual» el dato indexado corresponde a una muestra real—. Además, se extrae el valor de la muestra que es almacenada en la variable *y*. A este valor se le aplica la escala y desplazamiento vertical deseado y se corrige para que la gráfica se represente de abajo hacia arriba —líneas 28 y 29—. Si el dato indexado en el registro virtual no se corresponde con una muestra real —este índice no es divisible entre los puntos en pantalla que separan dos muestras—, se procede a tratar la siguiente coordenada horizontal de pantalla —variable *x*— para ver si el índice indexado del registro virtual corresponde con la posición de un dato en el registro real —línea 26—. Este proceso continua hasta que se ha barrido horizontalmente la pantalla. Si en la iteración del bucle se determina que se está tratando un dato real, se procede a dibujarlo. Esto se realiza en las sentencias entre las líneas 28 a 37, donde se discrimina si se representan solo las muestras reales —llamada a la función *ponPunto* en la línea 31— o si además se muestran los datos interpolados entre muestras —llamada a la función *línea* en la 34—. Para el caso de representar la línea que une dos muestras consecutivas, se necesita, además de las coordenadas de la muestra actual —valor de las variables *x* e *y*—, las de la muestra tratada con anterioridad guardadas en las variables *x_ant* e *y_ant*. La variable *y_ant* es realmente un atributo perteneciente a la clase Graph, por lo que entre llamada y llamada del método *draw* se conserva el valor almacenado. En cambio, la variable *x_ant* es una variable que existe solo en el ámbito de la función *draw*, por lo que para que conserve el valor entre llamadas a la misma, necesita que sea declarada como estática —línea 4—. Además, la declaración de esta variable como estática va a permitir que se puedan trazar líneas entre muestras consecutivas representadas en áreas invalidadas distintas.

Para determinar los datos interpolados en las zonas horizontales extremas del área invalidada, se utiliza el código existente entre las líneas 13 a 22. La condición de la línea 13 identifica si el dato a tratar está en estas zonas extremas, en tal caso se procede a interpolar el valor de estos datos, sino, se busca su valor en el registro real o no se realiza ninguna acción a la espera que el método *línea* realice la interpolación entre puntos como ya ha sido mencionado. Esta condición es finalmente comentada y sustituida por la que aparece en la línea siguiente —línea 14— que verifica lo mismo pero solo para toda la pantalla y no para cada área invalidada. Esto posibilita que se puedan visualizar las muestras adquiridas —cuando se manda visualizar solo puntos mediante la función *PonPunto*—, ya que si se hace la verificación para cada área, se representarían las muestras más los datos interpolados en los extremos de estas. En cambio para la representación donde se unen los puntos con líneas —mediante la función *línea*— es indiferente el cómo se haga. En cualquier caso, ya sea determinado los valores de los puntos extremos del área o de la pantalla, se necesita la interpolación de puntos. Esta interpolación está calculada entre las líneas 16 a 21. Para el mejor entendimiento de este proceso, se va a hacer uso del siguiente esquema:

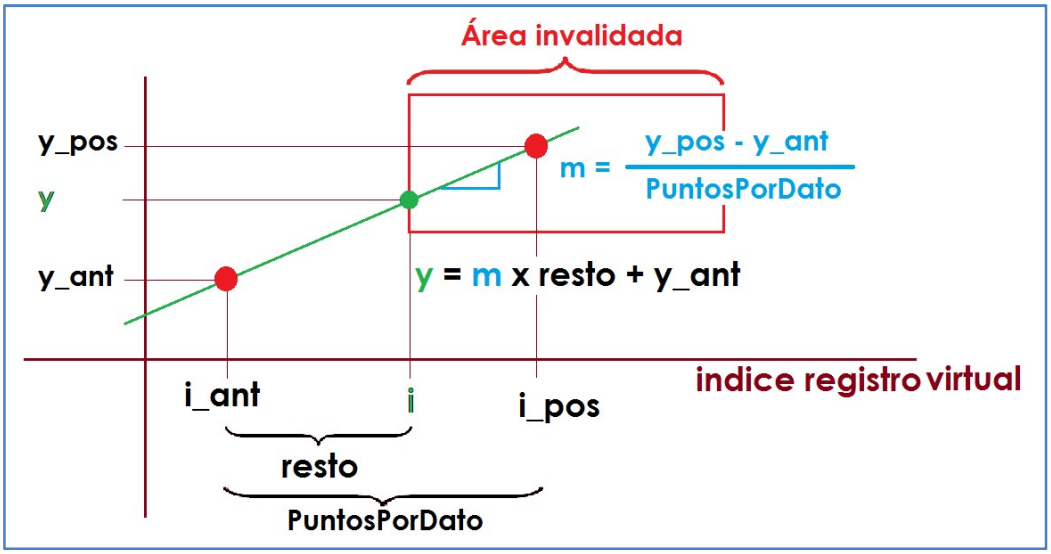


Ilustración 4.26: Interpolación de los puntos extremos del área invalidada o extremos de la pantalla

En este esquema se presentan las muestras reales como puntos rojos y el punto interpolado en el extremo del área invalidada —o de toda la pantalla, como es el caso— como un punto de color verde. Inicialmente se obtiene el índice del registro virtual del dato que corresponde al extremo del área invalidada mostrado en color verde como *i* en la ilustración. El valor de esta variable ha sido determinado en la línea 12 del código anterior y se ha verificado que pertenece al extremo del área invalidada en la condición de la línea 14. Es imprescindible calcular que valor interpolado tiene este punto —el valor aparece como *y* verde en la ilustración— ya que, si se presenta la gráfica uniendo los puntos con líneas —función *linea*—, se necesita este primer valor para trazar la primera línea —situación similar ocurre con el último punto y la última línea—. Para determinar este valor se necesita conocer el que tienen los puntos anterior y posterior que pertenecen al registro real —se representan como *y_ant* e *y_pos* en la ilustración—, y para ello es necesario conocer sus índices en este registro real. Se sabe que los índices del registro virtual que corresponde con los del registro real, son aquellos que son divisibles por la separación de puntos en pantalla entre dos muestras —que es el valor de la variable *PuntosPorDato*—, con lo que para determinar el índice anterior al índice que corresponde al borde del área invalidada en el registro virtual, se necesita restarle a este último índice —*i*— la distancia que le separa del índice anterior —*i_ant*—. Esta distancia es el resto resultado de dividir de forma entera el valor del índice en el borde del área invalidada entre la separación de puntos en pantalla entre muestras, es decir, entre el valor de la variable *PuntosPorDato* —mostrado en las líneas 16 y 17 del código anterior y en la ilustración mediante una llave que se para la distancia de *i* e *i_ant*—. Una vez obtenido *i_ant*, para determinar *i_pos* bastará con sumarle al primero el número de puntos que separan dos muestran en pantalla, esto es, el valor de la variable *PuntosPorDato* —mostrado en el código en la línea 18 y en la ilustración con una llave que separa ambas ordenadas—. Pero ambos índices, a pesar de hacer referencia a muestras reales, están expresados en términos del registro virtual. Para poder extraer el valor de las muestras, en el registro real, es necesario dividirlos por el valor *PuntosPorDato* ya que es la relación de escala entre ambos registros. Teniendo ya los valores de los índices anterior y posterior al índice del dato a interpolar —todos ellos referentes al registro virtual—, y pudiendo acceder a los valores de las muestras, se puede calcular la pendiente de la recta que unen las dos muestras, tal y como aparece en la ilustración en color azul y en el código entre las líneas 19 y 20 —en ambos casos la pendiente se denota con la letra *m*—. Finalmente, para calcular el valor interpolado hay que aplicar la expresión de la ecuación de la recta que aparece en la ilustración, bajo el área invalidada, y en el código en la línea 21.

Se ha visto como se accede a las muestras reales y como se determinan datos interpolados, pero quienes se encargan de mostrarlos en pantalla son los métodos *linea* y *ponPunto* de la clase *Graph*. Ambos necesitan conocer las coordenadas de lo que van a representar en forma absoluta, es decir, referentes a la región de

memoria bidimensional que representa la pantalla que se va a pintar a continuación: el buffer secundario de *TouchGFX*. La función *línea* está basada en la función *LCD_DrawUniLine* de la librería *STM32F429I-Discovery* —archivo stm32f429i_discovery_lcd.c que viene en la librería estándar de periféricos para la placa homónima— modificando su lista de parámetros y la llamada a la función encargada de dibujar realmente los puntos en pantalla; su código es:

```
1  void Graph::linea(int x1,int y1,int x2,int y2,uint8_t iS,Rect absoluteRect, Rect invalArea,
2                    uint16_t* frameBuffer)const
3  {
4    . . .
5     for (curpixel = 0; curpixel <= numpixels; curpixel++)
6      {
7          ponPunto(xpun,ypun, iS, absoluteRect, invalArea, frameBuffer);
8          . . .
9
10     }
11
12 }
```

Código 4.67: Método linea de la clase Graph

Necesita como argumentos, además de las coordenadas de los puntos a unir con una línea recta —coordenada del primer punto x1 e y1 y del segunda x2 e y2—, el orden de la serie que el objeto de tipo *Graph* está tratando —*iS*—, el rectángulo, expresado en coordenadas absolutas, que representa el objeto de tipo *Graph* —*absoluteRect*—, las coordenadas relativas del área invalidada —*invalArea*— y la dirección del buffer secundario que *TouchGFX* usará como buffer de pantalla en la siguiente actualización de la misma —*frameBuffer*—. Básicamente lo que hace esta función es determinar cuántos puntos y donde se van pintar entre los dos puntos a unir —algo ya implementado en la función *LCD_DrawUniLine*—, para a continuación llamar a la verdadera función que pinta sobre pantalla: la función *ponPunto*. Esta función necesita como argumentos la coordenada del punto a pintar, las coordenadas absolutas del componente —objeto tipo *Graph*—, las coordenadas relativas del área invalidada y la dirección del área de memoria a ser utilizado como buffer secundario bidimensional de pantalla. Estos tres últimos argumentos ya fueron pasados a la función *linea*, de hecho la función *linea* solo los necesita para pasárselos a la función *ponPunto*. El código de la función *ponPunto* es el siguiente:

```
1  void Graph::ponPunto(int xp, int yp, uint8_t iS, Rect absoluteRect, Rect invalArea,
2                       uint16_t* frameBuffer)const
3  {
4      int y = yp-(uint16_t)(Series[iS]->ancho_linea/2);
5      uint32_t offset;
6      offset = (absoluteRect.x + xp) + HAL::DISPLAY_WIDTH * (absoluteRect.y + y);
7      if (Series[iS]->Vectores && Series[iS]->TipoArea)
8          lineaArea(xp, yp, iS, absoluteRect, invalArea, frameBuffer);
9
10     for (int i=0; i<Series[iS]->ancho_linea; i++)
11     {
12         if ((y < (invalArea.y + invalArea.height) && y >= invalArea.y) )
13         {
14             frameBuffer[offset]=Series[iS]->Color;
15         }
16         y++;
17         offset=offset+HAL::DISPLAY_WIDTH;
18     }
19     . . .
20 }
```

Código 4.68: Función ponPunto de la clase Graph

Lo primero que se determina es la coordenada «*y*» —referida a la parte alta de la pantalla— donde comenzar a pintar el punto acorde con la anchura especificada para el mismo en el atributo *ancho_linea* de la serie que se está tratando —la especificada por el índice *iS* que identifica a la serie dentro del *array* de punteros a objetos de tipo serie («array» *Serie*) de la clase *Graph*—. Esta «anchura» se ha de pintar, verticalmente, a ambos lados del punto; por ello la coordenada vertical inicial para pintar está media anchura más alta de lo que

correspondería —línea 4—. A continuación se determina cual es la posición inicial del punto a pintar dentro del buffer secundario —dirección lineal en memoria—. Para ello se suma a la coordenada horizontal absoluta del control, la coordenada horizontal del punto a pintar que está referida al control —coordenada relativa— más tantos anchos de pantalla en memoria como indica la coordenada vertical del control más la coordenada vertical del punto —línea 6—. Esto queda más claro en la siguiente representación:

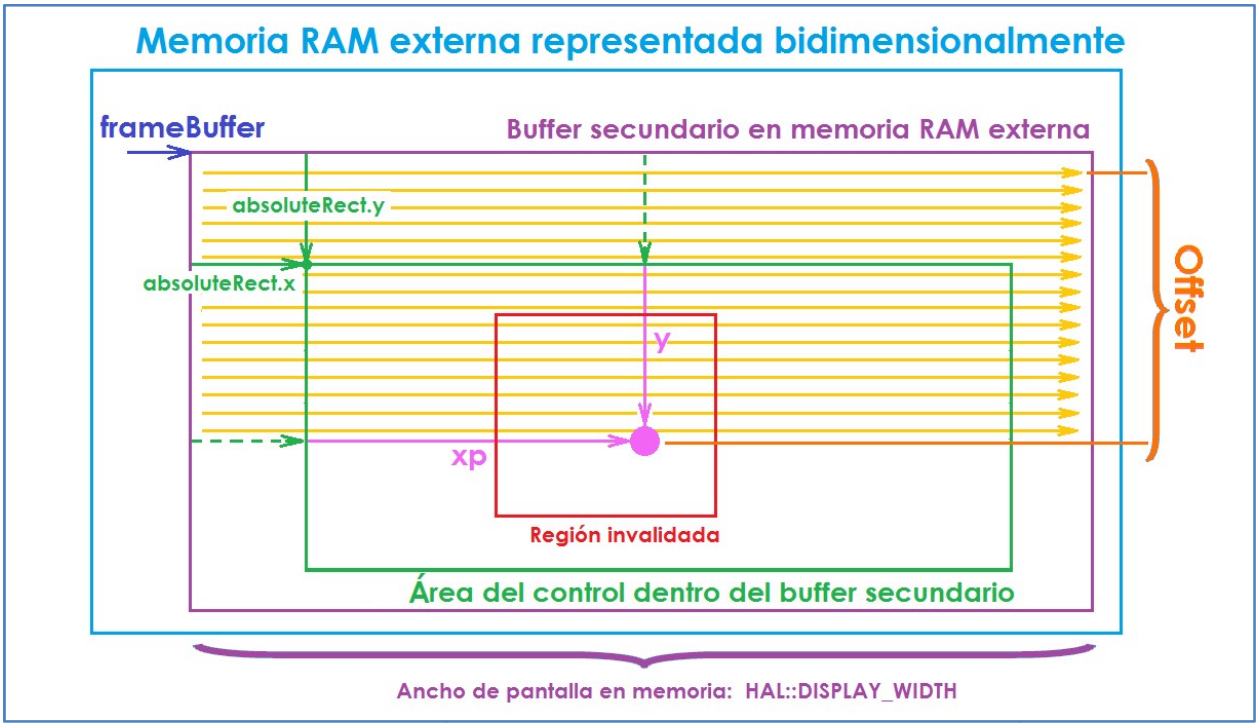


Ilustración 4.27: Representación de un punto en memoria dentro del área invalidada

En esta ilustración de aprecia que para llegar linealmente en memoria a la dirección del punto a pintar en pantalla —dibujado en color rosa—, se necesita recorrer tantos «anchos de pantalla en memoria» como indica la suma de las coordenadas verticales del control gráfico más la del punto —(*absoluteRect.y+y)*HAL::DISPLAY*, representado con los vectores amarillos—, para finalmente añadir lo que resta de memoria hasta llegar al punto — *absoluteRect.x + xp*—. Esto es calculado en la línea 6 del código anterior e igualado a la variable denominada *offset* —mostrado en la ilustración en color naranja—. Esta variable va a servir como índice del puntero *frameBuffer* que apunta a la región de memoria que representa la pantalla — recuadro de color violeta en la ilustración—. Este puntero ha sido declarado y definido dentro de la función *draw* y es de tipo puntero a entero sin signo de 16 bits —uint16_t*—, con lo que los datos en memoria se indexan de dos en dos bytes —que es el ancho del dato de cada punto en pantalla (RGB565) —. Sin embargo, el valor vertical del punto puede caer fuera del área invalidada, como se muestra en esta figura:

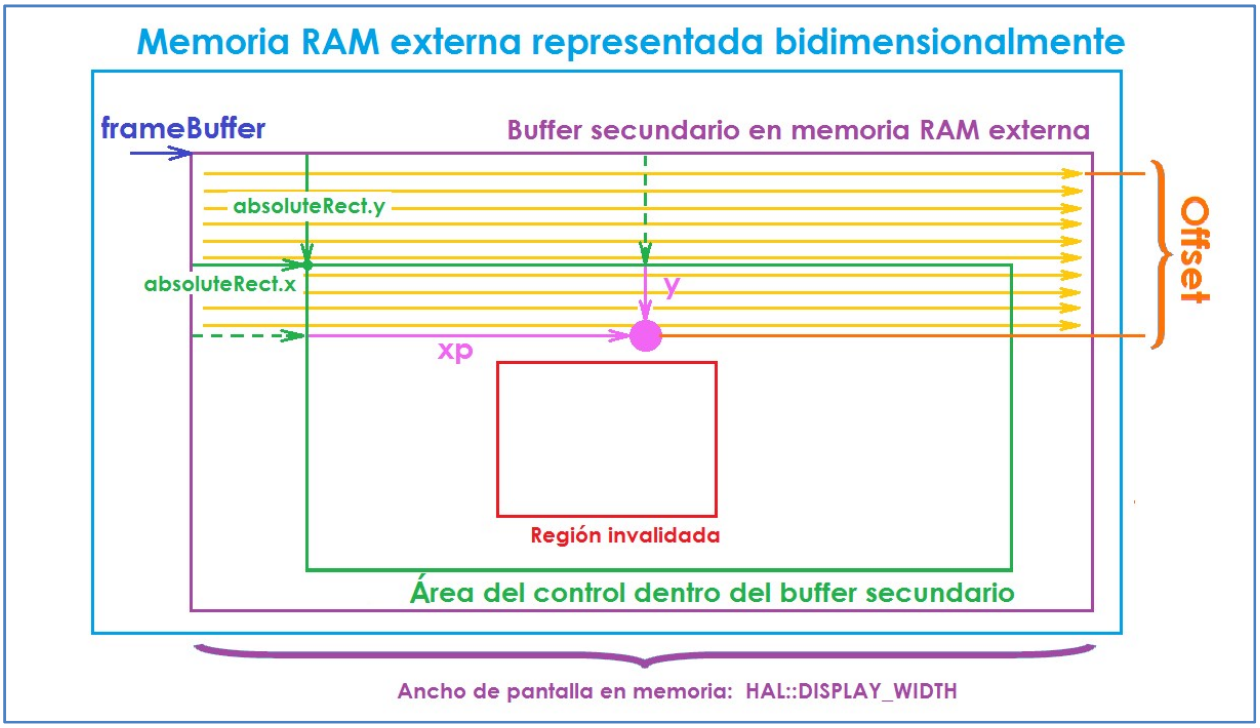


Ilustración 4.28: Representación de un punto en memoria fuera del área invalidada

El punto siempre va a caer dentro del rango horizontal del área invalidada, ya que, los puntos a mostrar en el área especificada de pantalla han sido indexados desde el registro teniendo en cuenta este rango horizontal, pero verticalmente solo depende del valor que tome cada dato dentro del mismo. Por ello es necesario comprobar si se produce tal situación y si es así, no proceder a dibujar el punto —no variar el valor de memoria que ocupa dicho punto— sino, se modificará una región que no ha sido solicitada por *TouchGFX* y el resultado puede ser inesperado. De ello se encarga la condición que aparece en la línea 12 del código. Pero la coordenada «*y*» ha sido modificada para que el punto se muestre con el grosor vertical deseado —atributo *ancho_linea* de la serie—, por ello se debe barrer en todo este ancho deseado verticalmente y ello se realiza entre las líneas 10 a 18. Se trata de un bucle *for* donde en cada iteración, previa comprobación que la coordenada vertical está dentro del área invalidada, se rellena la dirección de memoria correspondiente con el valor del color que se quiere que tenga el punto —el atributo *Color* de la serie que es invocado en la línea 14— y después se apunta a la coordenada vertical siguiente para ir completando este ancho vertical del punto —en las líneas 16 y 17—. Por último, en las líneas 7 y 8 esta la parte del código que se encarga de llamar a la función *lineaArea* si se quiere que la zona que se encuentra entre la representación de la serie y el eje horizontal quede relleno con un color o gama de colores. Esto solo se produce si es lo que se desea —atributo *TipoArea* de la serie con valor verdadero— y se están uniendo los puntos mediante líneas —atributo *Vectores* de la serie con valor verdadero—.

4.4.1.11.3 Esquema de color de las series.

El esquema de color de la serie es el relleno que potencialmente puede existir entre el borde de la gráfica y la parte inferior del control que la contiene —la abscisa—. Este relleno puede tratarse de un color solido o de un degradado o gama de color. En el caso de un color sólido, basta con un atributo que albergue el dato que represente este color, sin embargo, en el caso de la gama, se necesita toda un conjunto de datos que será más amplio cuanto más definición se desee en la transición de color. En este proyecto se opta por manejar gamas de colores en vez de un solo color, ya que proporciona una estética de profundidad y posibilita la creación futura de

un control que muestre el espectro en tipo «caída de agua» —donde en el eje «*x*» se muestre la frecuencia, en el «*y*» el tiempo, y el nivel se muestre en las distintas tonalidades de color—. Software matemático como *Matlab* u *Octave* manejan gamas de colores con distintas resoluciones para ser utilizadas en las representaciones de gráficas. Por ello, se va a aprovechar la capacidad de estos programas —en concreto *Octave*, ya que se trata de software gratuito—, para crear un *array* con la gama de color deseada mediante el uso de un script de *Octave* —con algunas pequeñas modificaciones también se puede usar en *Matlab*—. De todas las gamas de color que dispone Octave se van a utilizar siete: *Jet, Cool, Hot, Ocean, Rainbow, Winter, Autumn*. Estas gamas de color se muestran a continuación:

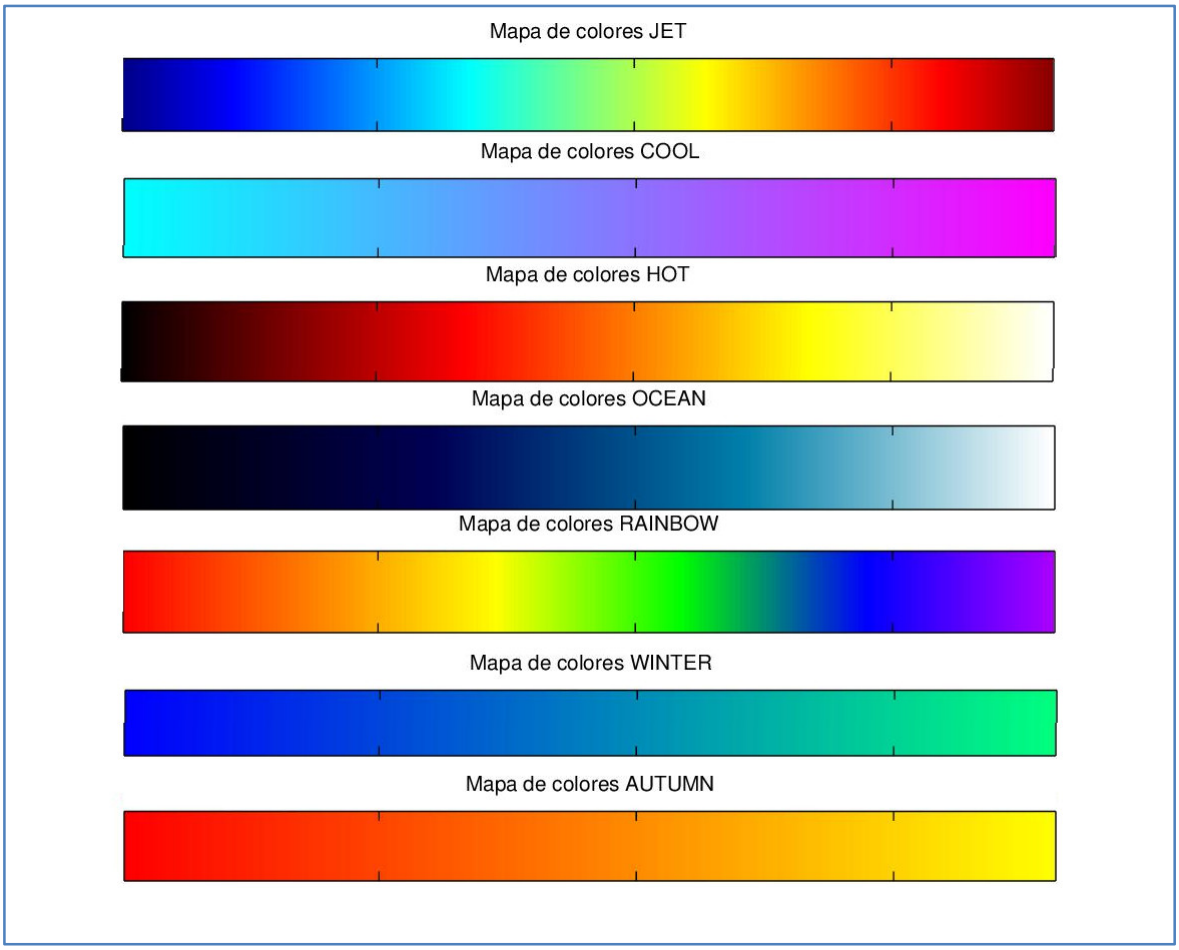


Ilustración 4.29: Gamas de color de relleno de las series

En el desarrollo final, en el menú de acceso rápido, se hace referencia a estas gamas como *Jet*, *Hielo*, *Fuego*, *Oceano*, *Arco Iris*, *Invierno* y *Verano*. El scripts para conseguir un *array* que contenga la información de todas estas gamas es el siguiente:

```
1  n=182;
2  n_gamas=7;
3  nombres={"jet","cool","hot","ocean","rainbow","winter","autumn"};
4
5
6  cadena_rgb565=sprintf('#include<gui/common/gamas.hpp>\n\n\nuint8_t n = N_DATOS_POR_GAMA;');
7  cadena_rgb565=strcat(cadena_rgb565,sprintf('\nuint16_t n_gamas = N_GAMAS;\n\n'));
8  cadena_rgb565=strcat(cadena_rgb565,sprintf('const uint16_t gamas[][N_DATOS_POR_GAMA]={'));
9
10 for j=1:n_gamas
11  comando=sprintf("%s(n)",nombres{j});
12  gama=eval(comando);
13  r=gama(:,1);
14  g=gama(:,2);
15  b=gama(:,3);
16  max_r=length(r);
17
18  cadena_rgb565=strcat(cadena_rgb565,sprintf('\n{\n'));
19  rgb565=zeros(1,max_r);
20  for i=1:max_r-1
21   rgb565(i)=bitshift(bitshift(uint16(r(i)*255),-3),11);
22   rgb565(i)=rgb565(i)+bitshift(bitshift(uint16(g(i)*255),-2),5);
23   rgb565(i)=rgb565(i)+bitshift(uint16(b(i)*255),-3);
24
25   if(mod((i),10)==0)
26     cadena_rgb565=strcat(cadena_rgb565,sprintf('0x%04x,\n',rgb565(i)));
27   else
28    cadena_rgb565=strcat(cadena_rgb565,sprintf('0x%04x,',rgb565(i)));
29   end
30  end
31  if(j~=n_gamas)
32   cadena_rgb565=strcat(cadena_rgb565,sprintf('0x%04x},\n',rgb565(max_r)));
33  else
34   cadena_rgb565=strcat(cadena_rgb565,sprintf('0x%04x}\n',rgb565(max_r)));
35  end
36 end
37 cadena_rgb565=strcat(cadena_rgb565,sprintf('};\n'));
38
39 fid = fopen ("gamas.cpp", "w");
40 fdisp (fid, cadena_rgb565);
41 fclose (fid);
42
43 fid2 = fopen ("gamas.hpp", "w");
44 cadena=sprintf('#ifndef GAMAS_H\n#define GAMAS_H\n\n#include <touchgfx/hal/HAL.hpp>\n');
45 fdisp (fid2, cadena);
46 cadena=sprintf('#define N_DATOS_POR_GAMA %d',n);
47 fdisp (fid2, cadena);
48 cadena=sprintf('#define N_GAMAS %d',n_gamas);
49 fdisp (fid2, cadena);
50 cadena=sprintf('\ntypedef enum TGamas\n{\n JET,\n COOL,\n HOT,\n OCEAN,');
51 fdisp (fid2, cadena);
52 cadena=sprintf(' RAINBOW,\n WINTER,\n AUTUMN\n}Gamas;\n');
53 fdisp (fid2, cadena);
54 cadena=sprintf('extern const uint16_t gamas[][N_DATOS_POR_GAMA];');
55 fdisp (fid2, cadena);
56 cadena=sprintf('extern uint8_t n;\nextern uint16_t n_gamas;\n\n#endif\n');
57 fdisp (fid2, cadena);
58 fclose (fid2);
```

Código 4.69: Script Octave para crear el array de gama de colores de las series

Este script va a generar el archivo *gamas.cpp* —líneas 39 a 41—, que contiene el *array gamas* que alberga las gamas de color, y el archivo *gamas.hpp* —líneas 43 a 58—, cabecera del archivo anterior y que contiene la declaración externa del *array gamas* —para que este pueda ser visto por otros archivos donde se incluya la cabecera—, la definición del tipo Gamas, y las declaraciones externas de las variables *n* y *n_gamas* —que representan el número de puntos por gama y el número de gamas, respectivamente, ambas declaradas en el archivo *gamas.cpp*—. Para la creación del *array* dentro del archivo *gamas.cpp*, en el script anterior se recorren cada una de las gamas a tratar —bucle *for* entre las líneas 10 al 36— para a continuación obtener los datos RGB de cada una de las gamas tratadas para una longitud de 182 datos —se obtiene por separado los datos de cada color elemental y son escritos en el archivo de gamas.cpp en formato hexadecimal mediante el bucle *for* interno existente entre las líneas 20 a 30—. El archivo *gamas.cpp* resultante es:

```
1  #include <gui/common/gamas.hpp>
2
3  uint8_t n = N_DATOS_POR_GAMA;
4  uint16_t n_gamas = N_GAMAS;
5
6  const uint16_t gamas[][N_DATOS_POR_GAMA]={
7  {
8  0x0010,0x0011,0x0012,0x0012,0x0013,0x0014,0x0014,0x0015,0x0016,0x0016,
9  . . .
10 0x9000,0x0000}, //gamma Jet
11
12 {
13 0x07ff,0x07ff,0x07ff,0x07df,0x07df,0x07df,0x0fbf,0x0fbf,0x0fbf,0x0f9f,
14 . . .
15 0xf81f,0x0000}, //gama Cool
16
17 {
18 0x0000,0x0800,0x0800,0x0800,0x1000,0x1000,0x1800,0x1800,0x2000,0x2000,
19 . . .
20 0xffff,0x0000}, //gamma Hot
21
22 {
23 0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0001,0x0001,0x0001,0x0001,
24 . . .
25 0xffff,0x0000}, //gama Ocean
26
27 {
28 0xf800,0xf820,0xf820,0xf840,0xf860,0xf880,0xf8a0,0xf8c0,0xf8e0,0xf900,
29 . . .
30 0xa01f,0x0000}, //gama Rainbow
31
32 {
33 0x001f,0x001f,0x001f,0x003f,0x003f,0x003f,0x005f,0x005f,0x005f,0x007f,
34 . . .
35 0x07f0,0x0000}, //gama Winter
36
37 {
38 0xf800,0xf800,0xf800,0xf820,0xf820,0xf820,0xf840,0xf840,0xf840,0xf860,
39 . . .
40 0xffe0,0x0000}  //gama Autumn
41 };
```

Código 4.70: Archivo fuente gamas.cpp

En las líneas 3 y 4 están definidas las variables que contienen el número de datos por gamas y el número disponibles de ellas respectivamente. Sus valores son constantes definidas en su archivo de cabecera —gui/common/gamas.hpp— y para este proyecto son, respectivamente, son 7 y 182. El *array* que contiene los datos —llamado gamas— se trata de un *array* bidimensional cuya primera dimensión indexa la gama a tratar —JET, COOL, HOT, OCEAN, RAINBOW, WINTER, AUTUMN— y la segunda, el dato concreto dentro de la gama. El archivo de cabecera generado es el siguiente:

```
1  #ifndef GAMAS_H
2  #define GAMAS_H
3
4  #include <touchgfx/hal/HAL.hpp>
5
6  #define N_DATOS_POR_GAMA 182
7  #define N_GAMAS 7
8
9  typedef enum TGamas
10 {
11  JET,
12  COOL,
13  HOT,
14  OCEAN,
15  RAINBOW,
16  WINTER,
17  AUTUMN
18 }Gamas;
19
20 extern const uint16_t gamas[][N_DATOS_POR_GAMA];
21 extern uint8_t n;
22 extern uint16_t n_gamas;
23
24 #endif
```

Código 4.71: Archivo de cabecera gamas.hpp

En él se definen las constantes que albergan los valores de número de datos por gama y número de gamas, antes comentados —líneas 6 y 7—. Se define el tipo enumerado *Gamas* entre las líneas 9 a 18 para ser utilizado como primer índice del *array gamas* del archivo fuente y seleccionar el esquema de color deseado. Finalmente se declaran como externas las variables *gamas*, *n* y *n_gamas* –líneas de la 20 a la 22— para poder ser accedidas desde aquellos archivos fuentes donde se incluya esta cabecera. Los archivos en donde está incluida esta cabecera directamente son *Graph.hpp* y *Model.hpp*. El archivo *Graph.hpp* necesita la inclusión de esta cabecera ya que contiene las clases *Graph-Serie* y estas necesitan acceder al contenido del *array gama,* ya que son las encargadas del dibujado de los esquemas. En el caso de *Model.hpp* contiene la clase del modelo encargada de mantener el estado de los atributos de todas las vistas cuando se carga una nueva y se destruye la anterior. Por ello contiene como atributo la variable *gamas* de tipo *Gamas* que guarda el esquema actual en uso; lo que obliga a incluir el archivo de cabecera mostrado en el código anterior. De forma indirecta esta cabecera estará incluida en todos aquellos lugares donde sea necesario incluir las cabeceras *Model.hpp* y *Graph.hpp*.

Para dibujar el esquema, tal y como fue comentado en el punto anterior —4.4.1.11.2—la clase *Graph*, al dibujar cada punto, y si la representación es de vectores y se desea mostrar el esquema de color bajo la serie —atributo *TipoArea* de la clase Serie— se llama al método *lineaArea* de la clase *Graph*, que parte de él es el siguiente:

```
1  void Graph::lineaArea(int16_t xp, int16_t yp, uint8_t iS, Rect absoluteRect,
2                        Rect invalArea, uint16_t* frameBuffer)const
3  {
4      DMA2D_InitTypeDef      DMA2D_InitStruct;
5      DMA2D_FG_InitTypeDef   DMA2D_FG_InitStruct;
6      DMA2D_BG_InitTypeDef   DMA2D_BG_InitStruct;
7
8      uint16_t longitud;
9      uint32_t add_dest;
10     uint32_t add_ori=(uint32_t)&gama;
11
12
13     if ((yp < (invalArea.y + invalArea.height) && yp >= invalArea.y)\
14          && (yp < rect.height-1 && yp > 0))
15     {
16         longitud=(invalArea.y+invalArea.height)-(yp);
17         add_ori=(uint32_t)&gama[0];
18     }
19     else if ((yp)<invalArea.y && (yp < rect.height-1 && yp >= 0))
20     {
21         longitud=invalArea.height;
22         add_ori=(uint32_t)&gama[invalArea.y-yp];
23         yp=invalArea.y;
24     }
25     else if ( yp < 0)
26     {
27         longitud=invalArea.height;
28         add_ori=(uint32_t)&gama[invalArea.y];
29         yp=invalArea.y;
30
31     }
32     else
33         return;
34
35     if (longitud>N_DATOS_POR_GAMA)longitud=N_DATOS_POR_GAMA;
36     . . .
37 }
```

Código 4.72: Indexación del array gamas en el método lineaArea de la clase Graph

Para saber que parte de la gama de color deseada se va mostrar en pantalla, se manejan las variables *longitud* y *add_ori* —líneas 8 y 10 respectivamente—. La primera de ellas va a contener la cantidad de datos del *array gamas* que será transferido en pantalla mientras que la segunda indicará a partir de qué dirección —índice— del mismo se inicia la transferencia. La variable *add_dest* —línea 9— contendrá el valor de la dirección del buffer secundario a donde transferir este fragmento del array mediante el uso directo del periférico DMA2D del micro STM32F4, pero esto será comentado más adelante cuando se muestre la parte de esta función donde se hace uso de ello.

En esta parte mostrada de la función, aparece un bloque *if - else if - else* —líneas de la 13 a la 33— que maneja las cuatro posibles situaciones. La primera de ellas se ilustra en la siguiente figura:

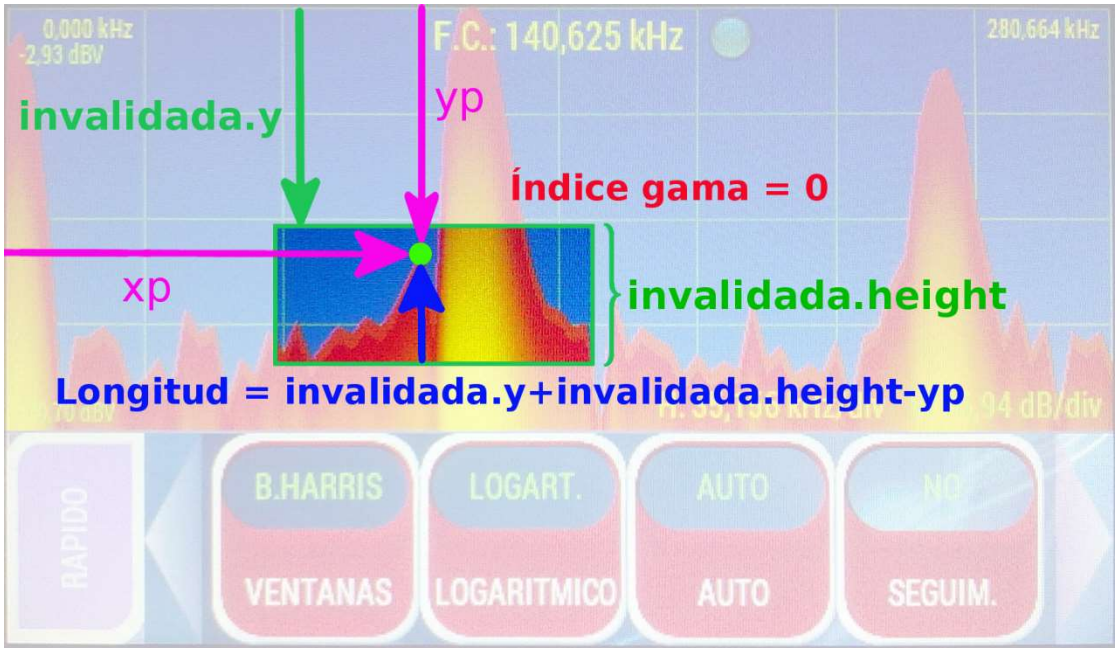


Ilustración 4.30: Dibujado del esquema desde el borde de la serie

En este caso, correspondiente a la condición que aparece en la línea 13, la coordenada horizontal tratada —que ha sido pasada como el primer argumento, *xp* a esta función (línea 1) — tiene su valor correspondiente del registro *yp* —segundo argumento de la función— dentro del área invalidada. Este punto es mostrado en color verde en la figura y se debe dibujar la gama de color seleccionada desde la posición vertical de este punto hasta el borde inferior del área invalidada. La zona de esta gama a dibujar —que realmente se trata de la línea vertical bajo el punto hasta la parte inferior del área invalidada— está representada como una flecha de color azul. Se trata de conocer el índice a partir del cual indexar el *array* gamas —valor de la variable *add_ori* actualizado en la línea 17— y la longitud de los puntos a tratar a partir de este índice —valor de la variable longitud actualizado en la línea 16—. El valor de este índice es cero –mostrado en texto rojo en la figura— ya que se va a dibujar la gama desde el borde de la serie. La longitud a trata del *array* está representada por la longitud de la flecha azul. Esta se calcula mediante los valores *invalidada.y*, *invalidada.height* e *yp* —mostrado como texto azul en la figura—; todos ellos obtenidos de los argumentos de la función.

La segunda situación se da cuando el valor de la serie —*yp*— está por encima del área invalidada pero cae dentro de la pantalla —condición mostrada en la línea 19—. Gráficamente esta situación es la mostrada en la siguiente figura:

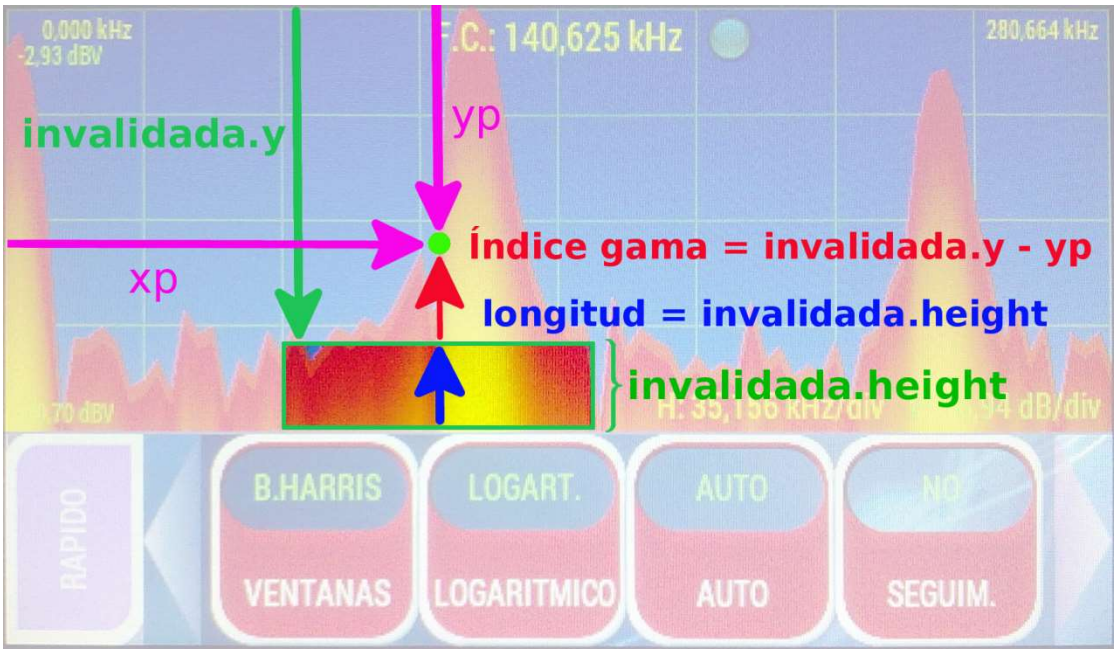


Ilustración 4.31: Dibujado de esquema por debajo del borde de la serie con punto dentro de pantalla

La longitud de los datos a tratar dentro del *array gamas* —flecha azul— es la altura del área invalidada —línea 21 del código—. El índice del *array* a partir del cual tratar los datos, es la distancia vertical que separa el punto de la serie de la parte superior de la zona invalidada —representada por la flecha de color rojo—. Mediante los argumentos pasados a la función y observando la figura, esta distancia se puede calcular como la diferencia entre la vertical del área invalidada y la de la serie —línea 22—. En la línea 23 se actualiza el valor de *yp* al de la vertical del área invalidad para posteriormente determinar la zona del buffer secundario a actualizar —se hablará de esto a continuación cuando se muestre la parte del código de la función que configura el DMA2D—. Esto se hace así ya que el valor de la vertical de la serie —*yp*— no va a ser utilizado más y se reutiliza la variable para este propósito.

La tercera situación es similar a la anterior con la diferencia que la vertical de la serie cae fuera de la pantalla y este valor ha sido saturado —punto verde de la siguiente figura—. En este caso los datos que se tratan del *array gamas* son a partir del índice cuyo valor es igual a la vertical del área invalidad —línea 28— y la longitud de los mismos corresponde a la altura de esta área —línea 27—. En la figura que se muestra a continuación, se presenta el caso particular en el que la vertical del área invalidada es cero:

Ilustración 4.32: Dibujado de esquema por debajo del borde de la serie con punto fuera de pantalla

Al igual que anteriormente, se reutiliza la variable *yp* para el mismo fin. Finalmente está la situación en la cual no se dan las anteriores, es decir, el caso en el que el área invalidada esté por encima de la serie, ocurrencia que no necesita tratamiento, ya que no se ha de dibujar ninguna gama, con lo que se retorna de la función —línea 33—. Existe una comprobación en la línea 35 que asegura que la longitud tratada del *array gamas* no sea más grande del tamaño de este. Si esto sucede, se limita la longitud al tamaño del *array*.

Una vez determinado el primer dato a tomar de la gama y a partir de él la longitud de datos a tratar, hay que transferirlos al buffer secundario en la zona del mismo que representa la zona de la pantalla a actualizar. Esto se hace con la parte del código que no se ha mostrado en la función *LineaAerea* que se encarga de configurar el periférico *DMA2D* y que se muestra a continuación:

```
1   add_dest=(uint32_t)&frameBuffer[((absoluteRect.x + xp) + \
2                                    HAL::DISPLAY_WIDTH * (absoluteRect.y + yp))];
3   DMA2D_DeInit();
4   DMA2D_InitStruct.DMA2D_Mode = DMA2D_M2M;
5   DMA2D_InitStruct.DMA2D_CMode = DMA2D_RGB565;
6   DMA2D_InitStruct.DMA2D_OutputMemoryAdd =  add_dest;
7   DMA2D_InitStruct.DMA2D_OutputGreen = 0;//Valor verde de fondo
8   DMA2D_InitStruct.DMA2D_OutputBlue = 0; // Valor azul de fondo
9   DMA2D_InitStruct.DMA2D_OutputRed = 0;  // Valor rojo de fondo
10  DMA2D_InitStruct.DMA2D_OutputAlpha = 0;
11  DMA2D_InitStruct.DMA2D_OutputOffset = HAL::DISPLAY_WIDTH-1;//this->getWidth() - 1;
12  DMA2D_InitStruct.DMA2D_NumberOfLine = longitud;
13  DMA2D_InitStruct.DMA2D_PixelPerLine = 1;
14  DMA2D_Init(&DMA2D_InitStruct);
15
16
17
18  DMA2D_FG_StructInit(&DMA2D_FG_InitStruct);
19  DMA2D_FG_InitStruct.DMA2D_FGCM = CM_RGB565;
20  DMA2D_FG_InitStruct.DMA2D_FGMA = add_ori
21  DMA2D_FG_InitStruct.DMA2D_FGO = 0;
22  DMA2D_FG_InitStruct.DMA2D_FGPFC_ALPHA_MODE = COMBINE_ALPHA_VALUE;
23  DMA2D_FG_InitStruct.DMA2D_FGPFC_ALPHA_VALUE = 100;
24  DMA2D_FGConfig(&DMA2D_FG_InitStruct);
25
26
27  DMA2D_BG_StructInit(&DMA2D_BG_InitStruct);
28  DMA2D_BG_InitStruct.DMA2D_BGCM = CM_RGB565;
29  DMA2D_BG_InitStruct.DMA2D_BGMA = add_dest;
30  DMA2D_BG_InitStruct.DMA2D_BGO = this->getWidth() - 1;
31  DMA2D_BG_InitStruct.DMA2D_BGPFC_ALPHA_MODE = NO_MODIF_ALPHA_VALUE;
32  DMA2D_BG_InitStruct.DMA2D_BGPFC_ALPHA_VALUE = 255;
33  DMA2D_BGConfig(&DMA2D_BG_InitStruct);
34
35
36  DMA2D_StartTransfer();
37
38  while (DMA2D_GetFlagStatus(DMA2D_FLAG_TC) == RESET);
```

Código 4.73: transferencia de parte del array gamas a pantalla del método lineaArea

Primero se determina la dirección inicial del área de memoria dentro del buffer secundario que representa la zona de la pantalla a actualizar. Esta se calcula —líneas 1 y 2— teniendo en cuenta las coordenadas del punto de inicio —*xp* e *yp*— y las coordenadas absolutas del control de tipo *Graph* tal y como ya se explicó en la sección 4.4.1.11.2. Luego se ha de configurar el periférico DMA2D del micro STM32F4 para realizar una «mezcla» especificando la región bidimensional de memoria de destino, donde se va a realizar, la región de origen de actuará de fondo y la región de origen que actuará de primer plano. La mezcla se realiza entre la región de fondo y la de primer plano —quedando la de primer plano por encima de la de fondo— y el resultado es almacenado en la región de destino. En este caso concreto se configura para que la región de destino y la de fondo sean la misma, es decir, se desea poner encima de la representación de la serie en memoria, la representación de la gama de color. Entre las líneas 3 a 14 se configura la región de destino, especificando su dirección de inicio —línea 6—, el espacio que deja la región destino a ambos en pantalla —configurado en la línea 11; de forma indirecta se está especificando la anchura de la pantalla (aunque para ello se necesita el siguiente parámetro a configurar: *PixelPerLine*) — y la altura de la región destino —configurado en la línea 12 con la longitud de los datos a transferir (longitud) —. Se ha de configurar también cuantos bytes es cada pixel —línea 5—, el tipo de mezcla que se va a realizar, que en este caso es la mezcla de un fondo y un primer plano sin tratamiento de mezcla alfa —línea 4— y cuantos pixeles de ancho tiene la región destino —línea 13—. Por otro lado, hay que configurar la región del fondo y primer plano de la mezcla especificando cuantos bytes tiene cada pixel de cada región —líneas 19 y 28 respectivamente—, las direcciones de inicio de cada una —líneas 20 y 29 respectivamente—, el espacio que dejan a ambos lados de la pantalla —líneas 21 y 30 respectivamente— y otras cuestiones de menos importancia ya que no se aplican en este tipo de mezcla como es la combinación alfa entre regiones y niveles alfa de cada una. Finalmente se realiza la mezcla —línea 36— y se espera hasta que haya concluido —línea 38—.

4.4.1.11.4 Utilización de la clase Graph-Serie dentro del proyecto.

En el archivo de cabecera de la única vista del proyecto —*FftView.hpp*— se declaran un objeto de tipo *Graph* —*grafico*—, un objeto de tipo *Serie* —*serieFFT*— y un contendor —*graphContainer*— que contenga al objeto de tipo *Graph* y otros varios objetos:

```
1  //*********************************************************************************
2   //Declaración de controles para el gráfico
3   //*********************************************************************************
4     Graph grafico;
5     Graph::Serie serieFFT;
6     Container graphContainer;
```

Código 4.74: Definición de objetos de tipo Graph y Serie

Y en el archivo fuente de la vista —*FftView.cpp*— se configuran estos objetos de la siguiente forma:

```
1  serieFFT.ponTipoDato(Graph::Serie::PUNTERO_8BITS);//superfluo, es así por defecto
2  serieFFT.ponDatos(registro, 1024);
3  serieFFT.ponVectores(true);
4  serieFFT.ponReferenciaVertical(0);
5  grafico.agregaSerie(serieFFT,false);
6  serieFFT.ponEscalaHor(0.2);
7  serieFFT.ponPosicionHor(-2320);
8  serieFFT.ponEscalaVert(0.7);
9  serieFFT.ponPosicionVert(0);
10 serieFFT.ponColor(gamas[RAINBOW][130]);
11 serieFFT.ponTipoArea(true);
12
13 graphContainer.setPosition(0,0,HAL::DISPLAY_WIDTH,HAL::DISPLAY_HEIGHT*2/3);
14 grafico.setPosition(0,0,HAL::DISPLAY_WIDTH,HAL::DISPLAY_HEIGHT*2/3);
15
16 graphContainer.add(FondoGrafico);
17 graphContainer.add(rejillaAnterior);
18 graphContainer.add(grafico);
19 graphContainer.add(rejillaPosterior);
20 graphContainer.invalidate();
```

Código 4.75: Configuración de los objeto de tipo Graph y Serie

Primeramente se configura el objeto serie, *serieFFT*, especificando que el tipo de datos que va a recibir es de 8 bits —línea 1—, que inicialmente accede a los primeros 1024 bytes —línea 2— de la región de memoria *registro* —un *array* de 4096 bytes—, que en la representación de la gráfica se van a unir los puntos con vectores —línea 3—, que la escala horizontal de la serie va a ser 1/5 —cada dato separado del siguiente por cinco puntos en pantalla (línea 6)—, que su desplazamiento será de -2320, lo que permite que el espectro se vea desde la frecuencia cero —línea 7—, que la escala vertical el gráfico será de 0,7 —solo a efectos de visualización en pantalla, ya que de la escala vertical real se hará cargo la aplicación dedicada a realizar la fft de forma concurrente con el proceso gráfico (línea 8) —, que no se añadirá desplazamiento vertical —línea 9—, que el color de la serie —no el esquema— será el valor del dato 130 dentro de la gama de color Arco Iris —línea 10— y que se dibujará la gama bajo la serie —línea 11—. La serie *serieFFT* es agregado a la gráfica *grafico* en la línea 5. Luego se configura el tamaño y posición tanto del contenedor del gráfico, *graphContainer*, como del objeto *grafico* en las respectivas líneas 13 y 14. Finalmente al objeto *graphContainer* se le añaden el objeto *FondoGráfico* —un objeto de tipo *Box* para generar un cuadro de color azul (el fondo del gráfico) —, el objeto *rejillaAnterior* —un objeto de tipo *Image* que contiene el dibujo de la rejilla con el resto de color transparente (para la rejilla del gráfico) —, el objeto *grafico* —objeto que contiene la serie *serieFFT*— y, de nuevo la rejilla: *rejillaPosterior*. El hecho de tener dos rejillas es debido a que al estar añadidas de esta forma al contenedor —primero la rejilla anterior, luego el gráfico y por último la rejilla posterior—, permiten que, mediante la puesta

a «invisible» de cada una de ellas de forma alternada, se pueda ver el gráfico delante o detrás del *grafico*. Para que se actualice el contenedor y todo lo que contiene, se llama a su método *invalídate* en la línea 20.

4.4.1.12 Componente gráfico «Cursor».

Se trata de un componente que señala el valor vertical de la serie especificada, representada en la gráfica, para un valor horizontal dado. Es un componente que trabaja muy estrechamente con el componente de representación de gráficas *Graph*. Como se comentó en la descripción del componente *Graph* —sección anterior, 4.4.1.11—, el componente Cursor, a pesar de tener que estar agregado de forma necesaria a un objeto *Graph*, no tiene su declaración de clase dentro de *Graph*, como sí sucede con la clase *Serie*. Esto es debido a que la creación de la clase Cursor es muy posterior a la clase *Graph* y esta última no necesita de la primera para funcionar, sin embargo, sí necesita de, al menos, una serie para hacerlo. La clase Cursor es creada por composición mediante la derivación de la clase *Container* —ver sección 1.13.2—. El contenedor consta de dos cajas —clase Box— de color sólido, distribuidas de tal forma que compongan una «cruz». La caja que constituye la línea vertical de esta cruz, está situada en una posición fija dentro del contenedor, moviéndose de forma solidaria con este, en cambio, la línea horizontal de la cruz se desplazará verticalmente a lo largo del contenedor. El esquema de construcción de un cursor es el siguiente:

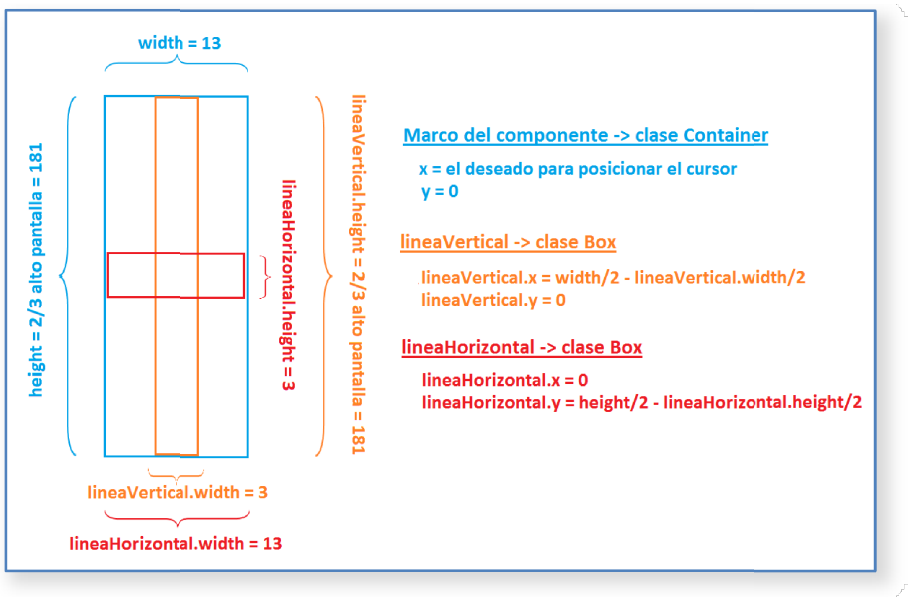


Ilustración 4.33: Composición de componente del Widget Cursor

Tal y como se muestra en esta ilustración, el valor inicial del componente —el contenedor—, es de dos tercios del alto de la pantalla y de trece puntos de ancho. Así mismo, la caja que representa la línea vertical tiene la misma altura que el contenedor que lo contiene, y su anchura es de tres puntos centrada sobre el contenedor. En cuanto a la caja que representa la línea horizontal, tiene la anchura del contenedor, es decir, trece puntos, dejando cinco puntos a cada lado de la línea vertical. La posición vertical, por defecto, de la línea horizontal, es centrada en el contenedor. Todos estos valores son fijados en el constructor de la clase, pero se pueden modificar algunos atributos; el ancho de las líneas puede variarse mediante el método *ponerAnchoLinea*, y la altura de la línea vertical —que corresponde con la de todo el componente— cuando se asocia el cursor a una

gráfica mediante el método *asociaGrafico* —aunque este método no es llamado directamente sino que lo llama el gráfico al registrar el cursor— La definición de la parte privada de la clase *Cursor* es la siguiente:

```
1  class Cursor : public Container
2   {
3     public:
4
5        typedef struct
6          {
7            int16_t ValorX;
8            int16_t ValorY;
9          }CoordenadasCursor;
10
11       typedef union
12         {
13           uint32_t ValorInt;
14           CoordenadasCursor Coordenada;
15         }CoordenadasEmpaquetadas;
16
17       . . .
18
19      private:
20
21       Box lineaVertical;
22       Box lineaHorizontal;
23       Graph *grafico;
24       CoordenadasEmpaquetadas argumentoRetrollamada;
25       GenericCallback<const Cursor &, CoordenadasEmpaquetadas >* cambioValorCallback;
26  };
```

Código 4.76: Definición de la clase «Serie»

En esta parte se aprecia que tiene como atributos una *lineaVertical* y *lineaHorizontal* de clase *Box*, un puntero al gráfico en el que está contenido, de tipo *Graph* —necesario para que el cursor adapte su altura a la del gráfico y tenga la posibilidad de solicitar que a este su actualización—, la coordenadas del cursor —en unidades de puntos de pantalla—, empaquetadas en el atributo *argumentoRetrollamada* y un objeto de tipo retrollamada —GenericCallback—, llamado *cambioValorCallback*, cuya misión es la de llamar al manipulador de usuario ante el acontecimiento —evento— de cambio de po*sición* del cursor. Es de señalar que las coordenadas son referentes al gráfico y son de tipo *CoordenadasEmpaquetadas* —líneas de la 11 a la 15—, una unión que permite tratar las coordenadas como un único valor empaquetado de 32 bits —el dato miembro *ValorInt*, en línea 13—, o como dos datos —dato miembro Coordenadas en línea 14, una estructura de tipo *CoordenadasCursor*, declarada entre la líneas 5 a 9—, el primero para la coordenada «x» y el segundo para la coordenada «y». El hecho de poder acceder a las coordenadas como un único valor de 32 bits o dos datos de de 16 bits, es debido a que por el objeto retrollamada solo puede enviar un único dato —el dato empaquetado—, mientras que el manipulador de usuario que obtenga este dato, tiene que ser capaz de desempaquetarlo para poder trabajar con cada coordenada de forma independiente.

Para la manipulación de la clase Cursor, se dispone, entre otros —ver código fuente—, de varios métodos. Están los métodos *ponerColor* y *ponerAlfa*, para modificar la apariencia del cursor —color y opacidad, respectivamente— y el método, ya mencionado, *ponerAnchoLinea*, para modificar la anchura del cursor. Para mover el cursor se dispone de los métodos *mueveX* y *actualizaValorVertical*. El primero, mueve horizontalmente todo el cursor en el valor —expresado en puntos de pantalla— del argumento pasado. La utilización de este método está destinada al usuario —es el usuario el que desea mover el cursor de una posición horizontal a otra— . El segundo método, en cambio, mueve la línea horizontal de cursor a lo largo de la altura de este —acorde con el valor del argumento pasado—. Este método no está pensado para utilización del usuario, sino que será el gráfico donde esté agregado quien lo llame.

Para la utilización del cursor junto con el gráfico, se han de seguir estos pasos:

1. Configuración estética del cursor, especificando su posición horizontal inicial —mediante el método *mueveX* —, su color —mediante el método *ponerColor*—, su opacidad —mediante el método *ponerAlfa*— y su visibilidad inicial —mediante el método *setVisible*—.

2. Registrar el cursor en el gráfico donde se localizará, mediante el método de este, *registraCursor*. El código de este método es el siguiente:

```
1  void Graph::registraCursor(Cursor &cursor)
2  {
3    nCursores++;
4    if (nCursores>MAXCURSORES) {nCursores--;return;}
5    Cursores[nCursores-1]=&cursor;
6    cursor.asociaGrafico(this);
7  }
```

Código 4.77: Código de la clase Graph para agregar un cursor

Primeramente se almacena el número de cursores registrados —línea 3—, solo si no se ha sobrepasado el límite de cursores registrables —línea 4—. Luego, se almacena la dirección del cursor en el array de cursores que la clase Graph posee —línea 5—. Finalmente, se pasa al cursor registrado la dirección del gráfico donde ha sido registrado. Esto se hace mediante el método *asociaGrafico* de la clase Cursor, de esta forma, al registrar el cursor, este adquiere la dirección del gráfico, sin necesidad de pasársela de forma explícita mediante la llamada a la función *asociaGrafico.*

3. Especificar sobre qué serie del gráfico estarán los cursores. Esto se consigue mediante la llamada a la función *ponSerieConCursores*, del objeto de tipo *Graph*, especificando en su argumento el índice de la serie sobre la que están los cursores —para saber cuántas series hay en el gráfico, se puede utilizar el método *obtenNSerie* del objeto de tipo *Graph*, que devuelve el número de ellas en el gráfico. Por defecto, la serie que recibe los cursores es la primera añadida al gráfico, por lo que, si solo se tiene una serie o es la primera la que debe recibir los cursores, no es necesario especificarlo.

4. Configurar un manipulador de usuario en el evento de cambio de valor vertical del cursor. Para ello es necesario, como en el resto de casos de creación de un manipulador de evento de usuario, crear un objeto de tipo Callback que tenga, como primer argumento, la clase de la vista que contenga el manipulador, como segundo, la referencia del objeto de tipo Cursor que dispara el evento, y como tercero, las coordenadas empaquetadas —de tipo *Cursor::CoordenadasEmpaquetadas*—. Luego hay que crear un método de esta vista, que sirva como manipulador y que tenga como primero argumento la referencia del objeto de tipo Cursor, y como segundo, estas coordenadas empaquetadas. Finalmente hay que construir el objeto Callback, especificando la dirección de la vista concreta que contiene el manipulador de usuario, y la dirección del propio método. Esto último se realiza en la lista de inicialización del constructor de la vista.

La actualización de los cursores se realiza desde dos posibles vías: desde el objeto de tipo Graph, al dibujar los puntos del gráfico, o desde los propios cursores que lo solicitan al gráfico. La actualización desde el objeto de tipo Graph se lleva a cabo desde su método encargado de dibujar los puntos del mismo. La porción de código de este método que lo realiza es el siguiente:

```
1  void Graph::ponPunto(int xp, int yp, uint8_t iS, . . .)const
2  {
3      . . .
4      if (iS==IndiceSerieConCursores)
5          actualizaCursor(xp, yp);
6  }
```

Código 4.78: Llamada de actualización de los cursores desde la clase Graph

Cuando el gráfico dibuja cada punto, comprueba si la serie, a quién pertenece ese punto, es sobre la que deben estar los cursores, y si es así, procede a actualizar el cursor mediante la llamada a su otro método *actualizaCursor*, pasándole las coordenadas de pantalla de ese punto. Este último método tiene el siguiente código:

```
1  void Graph::actualizaCursor(int xp, int yp)const
2  {
3      for (int i=0; i<nCursores; i++)
4      {
5          if (Cursores[i]!=0 && Cursores[i]->isVisible())
6          {
7              if (Cursores[i]->obtenX()==xp)
8                  Cursores[i]->actualizaValorVertical(yp);
9          }
10     }
11  }
```

Código 4.79: Método de la clase Graph para actualizar cursores

Primeramente se recorren todos los cursores registrados el objeto de tipo *Graph* —bucle *for* entre las líneas 3 a la 10—, luego, solo se procede a actualizar el cursor si realmente está registrado y si es visible —línea 5—. Finalmente, se llama al método *actualizaValorVertical* del cursor en cuestión, que manda actualizar la línea horizontal de la «cruz», que representa el cursor, con el valor vertical de la gráfica —línea 8—. Esta actualización solo se produce si el cursor está sobre la horizontal del punto que la gráfica está pintando en ese momento —línea 7—. La actualización de la posición de la línea horizontal del cursor se produce dentro de su método *actualizaValorVertical*:

```
1  void Cursor::actualizaValorVertical(int16_t y)
2  {
3    if((y+lineaHorizontal.getHeight()/2)>=this->getHeight())
4      {lineaHorizontal.setY(this->getHeight()-lineaHorizontal.getHeight());}
5    else if((y-lineaHorizontal.getHeight()/2)<=0)
6      {lineaHorizontal.setY(0);}
7    else
8      {lineaHorizontal.setY(y-lineaHorizontal.getHeight()/2);}
9
10
11   argumentoRetrollamada.Coordenada.ValorY=y;
12
13   if(cambioValorCallback && cambioValorCallback->isValid())
14     cambioValorCallback->execute(*this, argumentoRetrollamada);
15 }
```

Código 4.80: Método de actualización de la clase Cursor

El bloque if-esle_if-else, sirve para no dibujar el cursor fuera de los límites de la pantalla cuando el dato de la ordenada se encuentra fuera de ella —en la línea 3 y 4 se trata la situación en la que la ordenada está por abajo del gráfico (verticalmente), mientras que en las líneas 5 y 6, se trata la situación en la que la ordenada está por encima—. Si la ordenada se encuentra dentro del gráfico —líneas 7 y 8—, la línea horizontal del cursor se dibuja centrada sobre esta. Finalmente, este dato de ordenada es introducido en la variable de coordenadas empaquetadas con el fin de enviarlas al manipulador de evento de usuario. El dato de la abscisa ya había sido empaquetado al mover horizontalmente el cursor —mediante su método *mueveX*— a petición del usuario. Se recuerda que este último método es de utilidad exclusiva del usuario para que pueda mover el cursor horizontalmente donde le interese, lo que va a provocar todo el proceso de actualización de la señalización

vertical del este —mediante el posicionamiento de su línea horizontal tal y como se ha comentado—. La llamada al manipulador de evento de usuario solo se realiza si realmente se ha creado —línea 13—.

Como muestra final de código de la clase Cursor, se muestra el método *mueveX*, de utilidad de usuario:

```
1  void Cursor::mueveX(int16_t x)
2  {
3      setX(x-getWidth()/2);
4      argumentoRetrollamada.Coordenada.ValorX=x;
5  }
```

Código 4.81: Método mueveX de la clase Cursor

Tal y como ha sido mencionado, este método mueve el cursor horizontalmente a la coordenada «x» especificada, centrando la anchura del cursor sobre ella —línea 3—, y almacenando este abscisa en la variable que contiene las coordenadas empaquetadas.

La otra forma que se dispone para actualizar el cursor es a solicitud de este. Esto se realiza mediante la llamada a su método *solicitudActualizar*, donde lo que se hace es invalidar la gráfica para que de ese modo se desencadene todo el proceso anterior. Como ejemplo de su utilización, se muestra el manipulador de evento del elemento de menú llamado *elementoMenu3*, que sucede al finalizar la animación del mismo y que pertenece al código real de este proyecto:

```
1  void  FftView::Al_MenuAnimacionTerminada(const ElementoMenu &menu, bool estado)
2  {
3    if(&menu == &elementoMenu3)
4     {
5      Cursor1.setVisible(estado);
6      Cursor2.setVisible(estado);
7      Cursor1.solicitudActualizar();
8      Cursor2.solicitudActualizar();
9      actualizaTextoCursores();
10    }
11 }
```

Código 4.82: Ejemplo de utilización del método solicitaActualizar de la clase Cursor

El elemento de menú contiene, entre otros, controles para mover de forma horizontal dos cursores y muestras los datos de sus posiciones horizontales y verticales en la pantalla. Este manipulador de usuario se dispara cuando se ha mostrado u ocultado el elemento de menú tras su animación. Un dato que viene en su argumento —estado— señala si se ha mostrado el elemento de menú tras la animación —true— o si se ha ocultado —false—. Con esta información se muestra u oculta los cursores —líneas 5 y 6— y a continuación se utiliza la solicitud de actualización, que se comentaba, para ambos —líneas 7 y 8—. Luego se llama a una función que actualiza los textos en pantalla relativos a los cursores.

Como ejemplo de declaración de un cursor en el archivo de cabecera de la vista, se muestra el siguiente código:

```
1  class FftView : public View<FftPresenter>
2  {
3  public:
4      FftView():. . . ,
5               CursorCambioValorCallback(this, &FftView::Al_CursorCambioValor),
6                . . .
7               {. . .}
8          . . .
9  void Al_CursorCambioValor(const Cursor &cursor,\
10                           Cursor::CoordenadasEmpaquetadas coordenadas);
11      . . .
12 private:
13     . . .
14     Cursor Cursor1;
15          . . .
16         Callback<FftView, const Cursor &,\
17                 Cursor::CoordenadasEmpaquetadas > CursorCambioValorCallback;
18         . . .
19 };
20
```

Código 4.83: Declaración de un objeto de tipo Cursor

El objeto de tipo Cursor es declarado en la línea 14, mientras que el objeto de tipo Callback, necesario para llamar al manipulador de evento de usuario, se declara en las líneas 16 y 17. La construcción de este objeto es realizada en la lista de inicialización de la vista —línea 5—, pasándole la dirección del la vista —this— y la del método que actuará de manipulador de evento —*Al_CursorCambioValor*—, que es declarado en las líneas 9 y 10. En el archivo fuente de la vista se procedería a configurarlo:

```
1  void FftView::setupScreen()
2  {
3    . . .
4    Cursor1.mueveX(60);
5    Cursor1.ponerColor(Color::getColorFrom24BitRGB(0x00,0xff,0x00));
6    Cursor1.ponerAlfa(150);
7    Cursor1.setVisible(false);
8    grafico.registraCursor(Cursor1);
9    graphContainer.insert(&grafico,Cursor1);
10   Cursor1.ponerCambioValorCallback(CursorCambioValorCallback);
11   . . .
12 }
```

Código 4.84: Configuración de un objeto de tipo Cursor

Esta consiste en posicionarlo en el pixel 60 de pantalla —horizontalmente—, en la línea 4, luego, ponerlo de color verde —línea 5—, a continuación, poner una opacidad de 150 —línea 6—, e inicialmente que esté invisible. Para finalizar con su configuración, se registra en el gráfico, se añade al contenedor del gráfico justo encima de este, y se asigna el manipulador de evento de usuario.

Y el código del manipulador de evento de usuario tendría un aspecto similar al siguiente —es parte del código de este manipulador empleado en el desarrollo real—:

```
1  void FftView::Al_CursorCambioValor(const Cursor &cursor,\
2                                     Cursor::CoordenadasEmpaquetadas coordenadas)
3  {
4   static float x1=0, y1=0;
5   if (&cursor == &Cursor1)
6   {
7   x1=coordenadas.Coordenada.ValorX*this->presenter->obtenerModelo()->\
8                                       obtenerfrecuenciiaPorPixel()\
9             + this->presenter->obtenerModelo()->obtenerfrecuenciaMenorEnPantalla();
10           // se presenta el valor horizontal del cursor en Hz en pantalla
11
12  y1=-coordenadas.Coordenada.ValorY*this->presenter->obtenerModelo()->\
13                                         obtenerNivelPorPixelFFT()\
14            +this->presenter->obtenerModelo()->obtenerNivelMayorEnPantalla();
15          // se presenta el valor vertical del cursor en Hz en pantalla
16     }
17 }
```

Código 4.85: Definición del manipulador de evento al cambio de valor del Cursor

Al cambiar el valor del cursor —al cambiar la posición vertical de su línea horizontal—, se convierten sus coordenadas desde unidades de pantalla a Hercios, para la coordenada horizontal y en Voltios o dBV para la vertical.

4.4.2 Componentes gráficos adaptados procedentes de demostraciones.

4.4.2.1 Componente gráfico « HoldableButton»

Se trata de un componente de tipo botón —descendiente de la clase *Button*—, y tiene como particularidad generar eventos de forma continuada mientras está presionado, es decir que no solo se limita a ejecutar un código cuando se produce una presión o liberación sobre él, sino que ejecuta ese código de forma continuada mientras está presionado. Su aspecto en este proyecto es el siguiente:

Ilustración 4.34: Apariencia del control HoldableButton en este proyecto

Son controles que están en los submenús uno y tres del menú Controles. Los otros dos submenús —dos y cuatro—, a pesar de contener botones con apariencia idéntica a estos, no se tratan de *HoldableButtons*, sino de *Buttons*. Posee los siguientes atributos:

```
1  class HoldableButton : public touchgfx::Button
2  {
3      public:
4              . . .
5      private:
6          static const int startTicksBeforeContinuous = 10;
7          int ticksBeforeContinuous;
8          int ticks;
9          int ticksDesdeLaPresion;//JCC, número de ticks desde el inicio de la presión.
10 };
```

Código 4.86: Atributos de la clase HoldableButton

El miembro estático *startTicksBeforeContinuous*, se utiliza realmente como una constante y especifica el número de *ticks* —actualizaciones de pantalla— que deberán transcurrir para que, a partir de ese momento, se inicie la emisión repetida de eventos generados. Cuando se llegan a esos *ticks*, se produce el primer evento de

actualización, el siguiente se producirá en una unidad menos del valor especificado por *startTicksBeforeContinuous*, y así de forma sucesiva hasta llegar al valor cero, momento en el cual, se produce cada evento en cada *tick* del sistema —suponiendo que se mantiene presionado el control—. Desde el punto de vista del usuario, al mantener presionado el control, se experimenta un periodo inicial sin actividad y a partir de él, se producen eventos repetidos cuya cadencia se va acelerando hasta llegar a una cierta velocidad de repetición de eventos. Para controlar este proceso, la clase dispone del atributo *ticksBeforeContinuous*, que sirve para contener los *ticks* que quedan para generar el siguiente evento, y el atributo *ticks*, un contador que cuenta los *ticks* generados y que se comparan con *ticksBeforeContinuous* para determinar cuándo lanzar el siguiente evento. Cuando *ticksBeforeContinuous*, alcanza el valor de cero —el siguiente evento se lanza en el siguiente tick—, el contador *ticks* se mantiene a cero y ya no se incrementa. Para darle mayor funcionalidad a la clase, se añade el atributo *ticksDesdeLaPresion*, que es un contador, como el atributo *ticks*, con la diferencia que *ticksDesdeLaPresion*, siempre cuenta todos los ticks transcurridos mientras se presiona el control. La funcionalidad añadida se consigue cuando se gestiona el valor de este atributo en el manipulador de evento de usuario, posibilitando que se realice una acción determinada si se han alcanzado cierto número de eventos lanzados. Para ello, se provee el método *obtenticksDesdeLaPresion* para el acceso a este atributo.

Para que todo funcione, este control gestiona el evento *handleClickEven*, que reinicia los atributos cuando el control es presionado —poniendo el atributo *ticksBeforeContinuous* al valor del atributo estático *startTicksBeforeContinuous*, el atributos *ticks* a cero y *ticksDesdeLaPresion* a cero— y registrar el componente para recibir eventos de temporización —ver sección 1.11.3—. Cuando el control es aliviado de la presión, lo quita del registro de recepción de eventos de temporización. Por supuesto ha de gestionar el evento handleTickEvent para provocar la emisión de eventos de forma repetida, tal como ha sido descrito. Todo este funcionamiento queda plasmado en la siguiente porción de código donde se muestran los eventos *handleClickEvent* y *handleTickEvent*, que están definidos en la parte pública de la clase *HoldableButton*, y el método de acceso *obtenticksDesdeLaPresion*:

```
virtual void handleClickEvent(const touchgfx::ClickEvent& event)
 {
  Button::handleClickEvent(event);
  if(event.getType() == touchgfx::ClickEvent::PRESSED
   {
    ticks = 0;
    ticksBeforeContinuous = startTicksBeforeContinuous;
    ticksDesdeLaPresion = 0; //JCC
    touchgfx::Application::getInstance()->registerTimerWidget(this);
   }
  else
   {
    touchgfx::Application::getInstance()->unregisterTimerWidget(this);
        ticksDesdeLaPresion = 0; //JCC
   }
 }

virtual void handleTickEvent()
 {
  Button::handleTickEvent();
  if(pressed)
   {
    if(ticks == ticksBeforeContinuous)
    {
     ticksDesdeLaPresion++;//JCC
     if(action && action->isValid())
         action->execute(*this);
     ticks = 0;
     if(ticksBeforeContinuous)
         ticksBeforeContinuous--;
    }
    else
     ticks++;
   }
 }

int obtenticksDesdeLaPresion(void){return ticksDesdeLaPresion;}
```

Código 4.87: Gestión de eventos dentro de la clase HoldableButton

Entre las líneas 1 a 16 esta le evento *handleClickEvent*. Gestiona la presión con el reinicio de atributos y el registro de eventos de temporización entre las líneas 4 a 10, mientras que la liberación de la presión, con el borrado de eventos de temporización, se gestiona entre las líneas 11 a 15. El evento de temporización está entre las líneas 18 a 35. Entre las líneas 32 y 33 se gestiona los *ticks* para lanzar el siguiente evento, mientras que entre las líneas 23 a 31 el lanzamiento. Para su uso, se ha de declara el control dentro de la parte privada de la vista:

```
1  class FftView : public View<FftPresenter>
2  {
3    public:
4        FftView():BotonCallback(this, &FftView::Al_Boton){}
5        . . .
6        void Al_Boton(const AbstractButton &button);
7        . . .
8    private:
9        . . .
10       HoldableButton decFrecCentral;
11       . . .
12       Callback<FftView, const AbstractButton &> BotonCallback;
13 };
```

Código 4.88: Declaración del objeto de tipo HoldableButton

En esta porción de código también se muestra el objeto de tipo *Callback BotonCallback* —línea 12—, que servirá de nexo de unión entre el evento de pulsación y el manipulador de evento de usuario *Al_Boton* —línea 6—. La construcción del objeto *BotonCallback*, se realiza en la lista de inicialización del constructor de la vista —línea 4—.La configuración se realiza dentro del método *setupScreen* de la vista de una forma similar a la siguiente:

```
1  void FftView::setupScreen()
2  {
3    . . .
4    decFrecCentral.setBitmaps(Bitmap(BITMAP_DECREMENTA_ID), Bitmap(BITMAP_DECREMENTAP_ID));
5    decFrecCentral.setXY(65, contContr1.getHeight()/2-decFrecCentral.getHeight()/2);
6    decFrecCentral.setAction(BotonCallback);
7    . . .
8    add(decFrecCentral);
9  }
```

Código 4.89: Configuración del objeto de tipo HoldableButton

Se cargan las imágenes que servirán para mostrar el control pulsado y liberado, como un control de tipo *Button* ordinario —línea 4—, se posiciona dentro de la vista —línea 5—, se le asigna el objeto *BotonCallback*, que está enlazado con el método *Al_Boton*, para servir de manipulador de evento de usuario —línea 6—, y finalmente se añade a la vista —línea 8—.

Un ejemplo de manipulador de evento es el siguiente, donde el botón de tipo *HoldableButton* utiliza la mueva propiedad creada —*ticksDesdeLaPresion*— para controlar el incremento de velocidad de desplazamiento de una gráfica a medida que se mantenga presionado el control.

```
1  void FftView::Al_Boton(const AbstractButton &button)
2  {
3    if(&button==&decFrecCentral)
4    {
5     if (decFrecCentral.obtenticksDesdeLaPresion()<150)
6         serieFFT.decrementaPosH(1);
7     else if (decFrecCentral.obtenticksDesdeLaPresion()<300)
8         serieFFT.decrementaPosH(5);
9     else
10        serieFFT.decrementaPosH(10);
11     presenter->obtenerModelo()->ponerDespHor(serieFFT.obtenPosicionHor());
12   }
13 }
```

Código 4.90: Ejemplo de manipulador de evento para el objeto de tipo HoldableButton

4.4.2.2 Componente gráfico « SliderBar»

Es un componente que consiste en una barra sobre la que hay un botón que se desliza por ella al producirse el evento de presión o arrate. La barra tiene asociado unas cantidades numéricas en sus extremos y el botón que se desliza adquiere el valor proporcional a su posición. Para concretar qué valores maneja, se especifican las cantidades máxima y mínima de la barra y donde está el origen de la misma —donde se localiza la cantidad mínima—. Su orientación puede ser vertical u horizontal. Es un componente que no estaba disponible en la versión 4.2 de *TouchGFX*, pero sí a partir de la versión 4.3. Ya que se comienza el desarrollo en la versión 4.2 y afortunadamente existía este control en los ejemplos suministrados con la distribución, se emplea el código fuente de este componente al que se le practican una serie de modificaciones. Su aspecto en este proyecto es el siguiente:

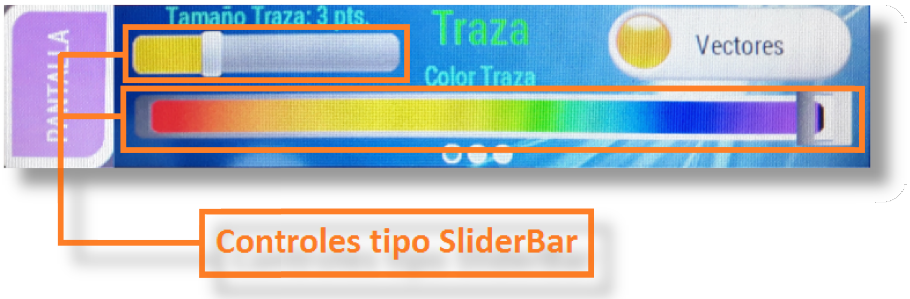


Ilustración 4.35: Apariencia del control SliderBar en este proyecto

Se trata de un componente cuyo código se reparte entre un archivo de cabecera —SliderBar.hpp— y un archivo fuente —SliderBar.cpp—. Las modificaciones realizadas consisten en la creación de dos tipos enumerados para controlar la orientación y el origen —lugar del valor mínimo—, así como los correspondientes atributos de dichos tipos y sus métodos de acceso; añadir características de transparencia al control; modificar los eventos *handleClickEvent* y *handleDragEvent* para que tengan en cuenta la orientación del control; y modificar la composición del control para que tenga la posibilidad de mostrar de forma diferente, el recorrido realizado por el botón deslizante de la barra. Su parte privada, así como la definición de tipos, es la siguiente:

```
1  class SliderBar : public Container
2  {
3      public:
4          . . .
5          typedef enum //enumeración creada
6          {
7              VERTICAL_ORIENTATION,
8              HORIZONTAL_ORIENTATION
9          } siliderOrientation;
10
11         typedef enum //enumeración creada
12         {
13             LEFT_UP_ORIGIN,
14             RIGHT_DOWN_ORIGIN
15         }origin;
16
17     private:
18         Image sliderBackground;
19         Image sliderBackgroundActive;
20         Image sliderKnop;
21         Container sliderBackgroundActiveViewPort;
22
23         uint16_t Orientation;//añadido
24         uint8_t Origin;//añadido
25
26         int16_t minLimit;
27         int16_t maxLimit;
28
29         uint8_t sliderRadius;
30
31         uint8_t Alpha;//añadido
32
33         GenericCallback<const SliderBar &, int16_t>* valueChangedCallback;
34         void setSlider(int16_t x, bool callback);
35 };
```

Código 4.91: Parte privada del control SlideBar

Entre la líneas 5 y 9 está la definición del tipo enumerado para controlar la orientación del control, y el atributo de la clase que alberga uno de los posibles valores de este tipo, aparece en la línea 23 —*Orientation*—. El tipo que establece dónde se localiza el valor mínimo de la barra, es igualmente un tipo enumerado cuya definición aparece entre las líneas 11 y 15. Sus posibles valores son LEFT_UP_ORIGIN, que estable el extremo izquierdo como valor mínimo, si la orientación del control es horizontal, o en el extremo superior si la orientación es vertical; y RIGHT_DOWN_ORIGIN, que establece el valor mínimo en el extremo derecho, si la orientación es horizontal o en el extremo inferior si la orientación es vertical. El atributo de la clase que aloja uno de los posibles valores de este tipo está en la línea 24 —*Origin*—. Para establecer los límites numéricos de la barra, están los atributos *minLimit* —valor menor de la barra, en línea 26— y *maxLimit* —mayor valor de la barra, en línea 27—. En el atributo *slideRadius* —línea 29— guardará el valor de la mitad de la anchura del botón que se desliza por la barra. Si este botón es circular, este valor coincide con su radio. Para permitir que el control tenga ciertos grados de transparencia, se crea el atributo *Alpha* —línea 31—. La parte privada de esta clase posee un método, *setSlider*, que es de uso interno del control para posicionar gráficamente el botón deslizable acorde con su valor pasado mediante el método público *setValue* —que es realmente el interfaz entre control y usuario para establecer el valor del botón deslizable—. No existe ningún atributo que albergue el valor del botón deslizable, queda implícitamente guardado mediante la posición relativa que tiene en la barra. Internamente el control hace las conversiones del valor a posición y viceversa en tantos por uno del intervalo eficaz que representa la longitud de la barra —restándole a la longitud de la barra la anchura del botón deslizable, ya que dicho botón no puede ir más allá de la longitud de la barra, y por tanto, la posición del medio de este botón jamás alcanzará los extremos de la barra, se queda a media anchura de él (si el botón es circular se puede decir que se queda a una distancia del extremo de la barra igual a su radio)—. Como atributo de tipo *Callback* —retrollamada—, posee *valueChangedCallback*, que se dispara —y ejecuta el método de la vista que sirve de manipulador de evento de usuario, si se ha realizado el enlace entre con este mediante el método de *SilderBar setValueChangedCallback* — si se establece un nuevo valor del botón deslizable mediante la llamada al método *setValue* o mediante el arrastre del botón sobre la barra o el toque de la barra en un lugar diferente al que se encuentra el botón deslizable. Por último, como atributos, posee lo correspondientes a su composición gráfica —líneas de la 18 a la

21—. Por ello, *SliderBar* es un control descendiente de *Container* —ver sección 1.13.2—. La imagen del botón deslizable quedará guardada en el atributo *sliderKnop* —línea 20—, la imagen de la barra en *sliderBackground* —línea 18—, y la imagen del recorrido del botón sobre la barra, desde su origen, en el atributo *sliderBackgroundActive*. Sin embargo para que esta imagen del recorrido del botón solo muestre la zona recorrida, debe estar contenida en un contenedor que varíe su longitud acorde con este recorrido. Este contendor es *sliderBackgroundActiveViewPort*. Para cargar todas estas imágenes, la clase *SliderBar* tiene el método público *setBitmaps*, cuyos argumentos, por este orden, son: la imagen del botón deslizable, la imagen de la barra y la imagen del recorrido del botón sobre la barra.

La declaración del objeto de tipo *SliderBar*, se realiza en la parte privada de la vista. En esta parte también debe estar el objeto de tipo *Callback* que, enlazado con un método de la vista a modo de manipulador de evento, lo ejecute en cada cambio del valor del objeto *SliderBar*:

```
1  class FftView : public View<FftPresenter>
2  {
3    public:
4      FftView():DeslizadorCambioValorCallback(this, &FftView::Al_DeslizadorCambioValor),
5                  . . ., {. . .}
6      void Al_DeslizadorCambioValor(const SliderBar &deslizador,int16_t valor);
7    private:
8      SliderBar deslizadorColorTraza;
9          . . .
10     Callback<FftView, const SliderBar &, int16_t>DeslizadorCambioValorCallback;
11     . . .
12 };
```

Código 4.92: Declaración del objeto SliderBar en la parte privada de la vista

En este código se muestra la declaración del objeto *deslizadorColorTraza* —línea 8— del tipo *SliderBar*, y la del objeto *DeslizadorCambioValorCallback* —línea 10— de tipo *Callback*, que ejecutará el método *Al_DeslizadorCambioValor* —línea 6— al producirse un cambio en el valor del objeto *SliderBar*. El enlace entre el objeto *Callback* y el método *Al_DeslizadorCambioValor* se realiza en el constructor de la vista —línea 4—. La configuración del objeto *deslizadorColorTraza* se realiza en el método *setupScreen* de la vista:

```
1  void FftView::setupScreen()
2  {
3    . . .
4    deslizadorColorTraza.setOrientation(SliderBar::HORIZONTAL_ORIENTATION);
5    deslizadorColorTraza.setBitmaps(Bitmap(BITMAP_BOTON_DESLIZADOR1_ID),
6                                    Bitmap(BITMAP_BARRA_DESLIZADOR1_ID),
7                                    Bitmap(BITMAP_BARRA_DESLIZADOR1_ID));
8    deslizadorColorTraza.setXY(10,botonActivacionVectores.getY()+\
9                               botonActivacionVectores.getHeight()+5);
10   deslizadorColorTraza.setLimits(0,181);
11   deslizadorColorTraza.setValue(181);
12   serieFFT.ponColor(gamas[RAINBOW][deslizadorColorTraza.getValue()]);
13   deslizadorColorTraza.setValueChangedCallback(DeslizadorCambioValorCallback);
14   contPant1.add(deslizadorColorTraza);
15   . . .
16 }
```

Código 4.93: Configuración del objeto SliderBar en el método setupScreen de la vista

Primero se ha de configurar su orientación, en este caso horizontal —línea 4— y su origen, que por defecto es LEFT_UP_ORIGIN y por tanto, no es necesario especificar. Luego, se han de establecer las imágenes usadas —líneas de la 5 a la 7— y su posición —líneas 8 y 9—. A continuación se establecen sus límites numéricos —línea 10— y el valor numérico de su botón deslizador —línea 11—, que será usado para determinar el color de la serie *serieFFT* —línea 12—. Finalmente, se configura el objeto *deslizadorColorTraza* para que al cambiar su valor ejecute el objeto de tipo *Callback* —línea 13—. El manipulador de evento puede tener el siguiente aspecto:

```
1  void FftView::Al_DeslizadorCambioValor(const SliderBar &deslizador,int16_t valor)
2  {
3    if (&deslizador == &deslizadorColorTraza)
4        serieFFT.ponColor(gamas[RAINBOW][valor]);
5  }
```

Código 4.94: Manipulador de evento disparado por el objeto SliderBar

Primeramente se comprueba cuál es el objeto que ha disparado el manipulador —línea 3— para poder ejecutar el bloque de código correcto. En este caso, al modificar el valor del objeto *deslizadorColorTraza*, se modifica el color de la serie *serieFFT* acorde con el valor del objeto *SliderBar* que indexa una tabla con colores.

4.4.2.3 Componente gráfico «JogWheel»

Es un componente facilitado por *TouchGFX* que no aparece como control estándar de la librería, sino que se facilita de forma indirecta a través de los códigos ejemplos, donde se muestra muy dependiente del resto del código de la vista a la que pertenece. Es por ello que se hace necesario practicarle una serie de modificaciones para hacerlo genérico y poder utilizarlo en cualquier tipo código.

Se trata de un control que permite seleccionar un valor al realizar giros sobre él. Al igual que el resto de controles, su apariencia dependerá de los bitmaps que se le suministren. En este proyecto se han utilizado dos controles modificados de este tipo, dentro del primer contenedor del menú Cursores. Su apariencia concreta, para este caso es:

Ilustración 4.36: Apariencia del control JogWheel en este proyecto

La parte privada de esta clase es la siguiente:

```
1  class JogWheel : public Container
2  {
3  public:
4  . . .
5  private:
6      Image background;
7      Image indicador;
8
9      Bitmap backgroundImage;
10     Bitmap backgroundImageWhenDragged;
11     Bitmap ImagenIndicador;
12
13     int32_t degreesRotated;
14     int16_t startValue;
15     int16_t deltaValueForOneRound;
16     int16_t currentValue;
17
18     bool firstDragEvent;
19     int16_t centerX;
20     int16_t centerY;
21     int16_t radius;
22     int16_t oldX;
23     int16_t oldY;
24     double oldLength;
25     const uint16_t ANGLE_MULTIPLIER;
26
27     GenericCallback<const JogWheel &, int16_t >* valueChangedCallback;
28     GenericCallback<const JogWheel &, int16_t >* endDragEventCallback;
29 };
```

Código 4.95: Parte privada de la clase JogWheel

Posee un control de tipo *image*, *background* —línea 6—, que contendrá la imagen del control en estado normal —*backgroundImage*, en línea 9— u opcionalmente, la imagen cuando está presionado —*backgroundImageWhenDragged*, en línea 10—. Como una de las modificaciones hecha, se le agrega el control de tipo *image*, *indicador* —línea 7— que se carga con el *bitmap ImagenIndicador* —línea 11—, cuyo cometido es moverse de forma solidaria con el giro practicado en el control para indicar, gráficamente, que efectivamente se está produciendo el giro. Este indicador puede ser cualquier *bitmap*, pero en este caso, se utiliza uno que se asemeja a la típica muesca que tienen los botones giratorios para que el usuario introduzca el dedo. Este control devuelve, a través de eventos, un valor que es proporcional al ángulo de giro practicado. Para la configuración de este valor devuelto, se utilizan los atributos *startValue* —línea 14—, para establecer el valor inicial, y *deltaValueForOneRound* —línea 15—, valor que determina qué valor, por cada vuelta girada, se añade al valor inicial. Internamente, el control utiliza los atributos *currentValue*, —línea 16—, que es la suma del valor inicial más el valor que representa el ángulo girado y *degreesRotated* —línea 13—, que representa el valor del ángulo girado. Para traducir este ángulo girado al valor devuelto por los eventos, se hace uso interno de más atributos. Así, *centerX* y *centerY* —líneas 19 y 20—, son las coordenadas del centro del control, que junto con los datos de las coordenadas del evento de arrastre —*DraggEvent*— que internamente gestiona el control, se determina las coordenadas del vector que une el centro con la posición de la presión realizada y que están contenidas en los unas variables interna llamadas *vx* y *vy*. En cada arrastre —cada *tick*— el valor de estas coordenadas pasa a los atributos *oldX* y *oldY* —líneas 22 y 23—, representando el vector calculado con anterioridad. Con los valores del vector actual —*vx* y *vy*— y el calculado con anterioridad — *oldX* y *oldY* —, se calcula, mediante el producto escalar de vectores —producto punto—, el ángulo girado —en radianes— y mediante el producto vectorial —producto aspa—, el sentido del ángulo de giro —su signo—. Con estos datos y el valor almacenado en *deltaValueForOneRound*, se calcula que valor se ha de añadir a *startValue*, y si esta suma es diferente al valor actual — *currentValue* —, se actualiza el valor actual y se dispara el evento que posibilita recuperar este valor. Los eventos manejados son *valueChangedCallback* y *endDragEventCallback* —líneas 27 y 28—, el primero se dispara al producirse el arrastre —giro— sobre el control en cada *tick*, mientras que el segundo se dispara al

liberar la presión sobre el control. Ambos tienen dos parámetros, el primero es la referencia del control que dispara el evento. Esto constituye otra modificación practicada sobre el control, ya que originariamente no lo poseía y no podía utilizarse un mismo manipulador de evento para distintos controles de tipo *JogWheel*. El segundo argumento es el valor que representa el giro efectuado en el control.

Un ejemplo de utilización de este control en el archivo de cabecera de la vista es el siguiente:

```
1  class FftView : public View<FftPresenter>
2  {
3  public:
4      FftView():
5          . . .
6          RuedaCursor1(60,24),
7          RuedaGiroCallback(this, &FftView::Al_RuedaGiro),
8          . . .
9
10     void Al_RuedaGiro(const JogWheel &rueda, int16_t valor);
11
12 private:
13         JogWheel RuedaCursor1;
14         Callback<FftView, const JogWheel &, int16_t > RuedaGiroCallback;
15          . . .
16 };
```

Código 4.96: Utilización de la clase JogWheel en el archivo de cabecera de la vista

Se ha de utilizar su constructor en el de la vista —línea 6—, pasándole como argumentos el valor inicial y el valor que añadir al anterior en cada vuelta girada. En el constructor de la vista también se ha inicializar el objeto *Callback* —línea 7—, pasándole como argumentos la dirección de la vista que lo contiene y la dirección del método de la vista que actuará como manipulador de evento —línea 10—. Tanto el objeto de tipo *JogWheel* como el objeto *Callback* se ha de declarar en la parte privada de la vista —líneas 13 y 14—.

Luego, se configura el objeto de tipo JogWheel y define el manipulador de evento en la archivo fuente de la vista:

```
1  void FftView::setupScreen()
2  {
3   RuedaCursor1.setBitmaps(Bitmap(BITMAP_RUEDA_ID),Bitmap(BITMAP_RUEDA_ID));
4   RuedaCursor1.setXY(5,0);
5   RuedaCursor1.setValueUpdatedCallback(RuedaGiroCallback);
6   RuedaCursor1.setIndicador(Bitmap(BITMAP_INDICADORRUEDA_ID));
7   contCur1.add(RuedaCursor1);
8  }
9
10 void FftView::Al_RuedaGiro(const JogWheel &rueda, int16_t valor)
11 {
12  if(valor<480 && valor>=0)
13   {
14    if(&rueda == &RuedaCursor1)
15     {
16      Cursor1.mueveX(valor);
17     }
18   }
19  }
```

Código 4.97: Utilización de la clase JogWheel en el archivo fuente de la vista

En el método *setupScreen* de la vista se configura el objeto de tipo *JogWheel*, estableciendo los *bitmaps* que usará para su aspecto en estado de reposo y presionado —en línea 3, en este caso, ambos son el mismo—, determinar su posición en la vista —línea 4—, cargar el objeto *callback* que va a llamar al manipulador del evento —línea 5—, poner el bitmap que indicará la posición de giro del control —línea 6—, y finalmente, añadir el control a la vista —línea 7—. El manipulador de evento —líneas de la 10 a la 19— será llamado en cada *tick* con el valor del ángulo de giro recorrido. En este caso, se comprueba que el valor que devuelve el control no exceda el

valor de 480 ni sea menor a 0—línea 12—. Como en cualquier manipulador de evento, se ha de verificar qué objeto ha producido el evento —línea 14— y ejecutar el bloque de código correspondiente, en este caso, se ejecuta el único mostrado, que está entre las líneas 15 a 17 y que tiene como misión transmitir el valor que viene del objeto de tipo *JogWheel* a otro control gráfico: el cursor Cursor1.

4.5 Desarrollo de la aplicación: clase DispositivoFFT.

El desarrollo de la aplicación en sí, está encapsulada en la clase *DispositivoFFT*. El objetivo de la misma, básicamente, es realizar la captura de los datos temporales y calcular la transformada discreta de Fourier. También realiza otras acciones accesorias como la aplicación de distintas ventanas temporales a los datos adquiridos, el cálculo del logaritmo del espectro y el escalado gráfico de los datos procesados para ser presentados por pantalla.

El método más importante de esta clase es *tareaFFT*, una función miembro que será ejecutada de forma indefinida dentro de una tarea del sistema RTOS —*FreeRTOS*—. Su meta es realizar la adquisición y los cálculos sobre los datos acorde con las órdenes recibidas desde el entorno gráfico, construido sobre la librería *TouchGFX*. Ya que el entorno gráfico se ejecuta en otra tarea RTOS, la única forma de comunicación entre este y la aplicación —*DispositivoFFT*—, es la utilización de colas de sistema operativo. El siguiente esquema muestra la relación entre un objeto de tipo *DispositivoFFT* y el entorno gráfico:

Ilustración 4.37: Detalle de la comunicación de la aplicación y el entorno gráfico

En esta ilustración aparece la parte del entorno gráfica oscurecida, ya que será tratada en el apartado 4.6. Por el momento solo interesa cómo la tarea de la aplicación se comporta en relación a la comunicación con la tarea gráfica, sin entrar en detalle en lo que ocurre dentro de esta última.

El método *tareaFFT* recibe de forma periódica mensajes a través de la cola de sistema desde el entorno gráfico —flecha bidireccional de color azul— que le indica que debe hacer, es decir, qué se le está solicitando —podría entenderse estos mensajes como órdenes desde el contexto gráfico—. Estas órdenes van desde la

solicitud de datos hasta la forma en la que se deben manipular los mismos —ver apartados 4.5.8, 4.5.9 y 4.5.10—. Para la ejecución de estas órdenes la *tareaFFT* hace uso de otro método de la clase *DispositivoFFT*: *interpretaMensaje*. Este último, interpreta el mensaje entrante por la cola que *tareaFFT* ha recibido. Se trata de un bloque de tipo *switch-case* donde se modificarán los atributos oportunos, acorde con el mensaje, para que la *tareaFFT* actúe en consecuencia. Por tanto, la comunicación dentro del contexto de la aplicación consiste, por un lado, en la llamada a la función *intrerpretaMensaje* —comunicación de *tareaFFT* hacia *interpretaMensaje*—, y por otro, en la modificación de atributos por parte de esta última —comunicación de *interpretaMensaje* hacia *tareaFFT*—.

Ya que para el cálculo de la DTF, el objeto de tipo *DispositivoFFT* va a hacer uso de la librería CMSIS-DSP, que la implementa a través del algoritmo FFT, es necesario revisar conceptos previos como son dicho algoritmo —en concreto el diezmado en frecuencia— así como las estructuras y funciones que usa esta librería.

4.5.1 Algoritmo FFT [12] [19] [18]

El algoritmo FFT es en realidad un grupo de algoritmos destinados a realizar el cálculo de la Transformada Discreta de Fourier de forma eficiente, minimizando las operaciones necesarias para su desarrollo. En este proyecto se han utilizado las funciones DSP CMSIS provistas para tal fin. El algoritmo FFT concreto utilizado en esta librería es el conocido como «diezmado en frecuencia», es por ello que será el único a tratar en este apartado. Aunque la implementación que hace CMSIS de este algoritmo es un poco más compleja de la presentada aquí, conceptos necesarios para su utilización como son los «factores de rotación» o «inversión de bits» son equivalentes.

Se definen la transformada discreta de Fourier y la transformada inversa como:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, e^{-j\frac{2\pi}{N}nk}, \qquad k = 0, 1, \dots, N-1 \tag{4.23}$$

$$x[n] = \frac{1}{N}\sum_{n=0}^{N-1} X[k]\, e^{j\frac{2\pi}{N}nk}, \qquad n = 0, 1, \dots, N-1 \tag{4.24}$$

Donde *n* es el número de orden de la muestra en el dominio del tiempo —tiempo discreto—, *k*, el índice del dato en el dominio discreto de frecuencia —frecuencia discreta—, *N*, el número de datos del registro, *x[n]*, es el dato en el dominio discreto de tiempo y *X[k]*, el dato en el dominio de la frecuencia discreta. En la primera expresión se muestra la transformada directa mientras que en la segunda, la transformada inversa. Ambas son prácticamente iguales con la salvedad del factor *1/N* y el signo del exponente de la exponencial compleja de la transformada inversa respecto de la directa. Por ello CMSIS va a utilizar las mismas funciones para el cálculo de una u otra con la salvedad del cambio de valor en algún argumento. A causa de esto, la próxima discusión se centrará exclusivamente en la expresión de la transformada directa. Para mayor sencillez en las expresiones, se denota la exponencial compleja de la transformada como:

$$W_N^{nk} = e^{-j\frac{2\pi}{N}nk} = cos\left(\frac{2\pi}{N}nk\right) - j\, sen\left(\frac{2\pi}{N}nk\right) \tag{4.25}$$

Quedando la transformada como:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{nk} \qquad (4.26)$$

Donde W_N^{nk} es conocido como «Twiddle factor» y podría traducirse como «factor de rotación». Esta denominación puede ser debida a sus propiedades de periodicidad y simetría que hacen que se repitan cada *N* elementos. Por ejemplo, para *N* igual a ocho y k constante —de valor uno—, el esquema de estos factores de rotación sería el siguiente [15]:

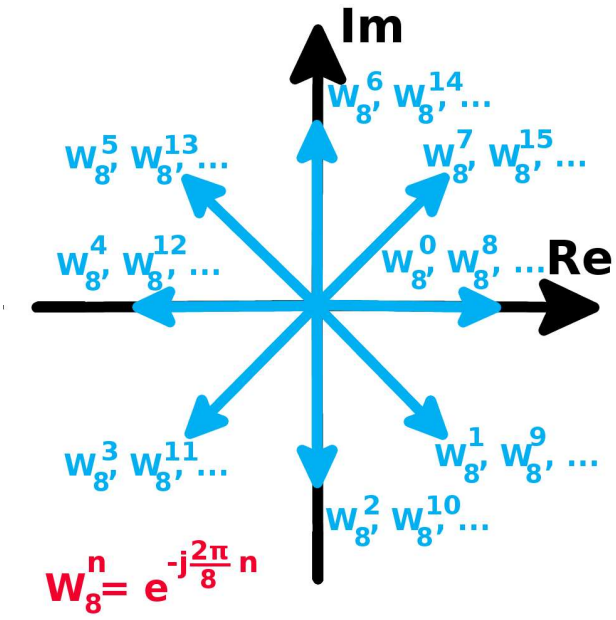


Ilustración 4.38: Representación de los factores de rotación —«twiddle factors»—

En esta ilustración se muestra como el factor de rotación cero, es igual al ocho, igual al dieciséis, etcétera. Cada factor de rotación es igual a aquel que está a una distancia de N, que corresponde a una vuelta y que en este caso es de ocho. Esta característica es la conocida como propiedad de periodicidad que matemáticamente será expuesta más adelante. También se cumple que cada factor de rotación es igual al opuesto de aquel que está a una distancia de N/2, que corresponde a media vuelta y que en este caso es de cuatro. Así, el factor de rotación uno, es igual al opuesto del cinco. Esta es la propiedad de simetría que se expondrá después.

Si se implementa el cálculo de la transformada mediante el desarrollo de la expresión anterior —(4.26) más atrás—, se tiene que:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{nk} = \; x[0]W_N^0 + x[1]W_N^k, + x[2]W_N^{2k} + \cdots + + x[N-1]W_N^{(N-1)k}\,, \qquad (4.27)$$
$$k = 0, 1, \ldots, N-1$$

Lo que muestra que para un valor concreto de *k* —frecuencia discreta—, se necesitan *N* multiplicaciones complejas y *N-1* sumas complejas. Teniendo en cuenta que *k* va desde *0* a *N-1* —*N* valores distintos de *k*—, se tienen en total N^2 multiplicaciones complejas y *N(N-1)* sumas complejas. Sin embargo el cálculo en una máquina

se elabora sobre números reales[22]. Expresando la fórmula (4.26) más atrás, en notación compleja cartesiana queda como:

$$X[k] = \sum_{n=0}^{N-1} \Big[\Big(Re\{x[n]\} Re\{W_N^{nk}\} - Im\{x[n]\} Im\{W_N^{nk}\} \Big) + j\Big(Re\{x[n]\} Im\{W_N^{nk}\} + Im\{x[n]\} Re\{W_N^{nk}\} \Big) \Big], \qquad k = 0, 1, \dots, N-1 \tag{4.28}$$

Esto supone que para cada sumando desarrollado en la expresión (4.27) más atrás, se tienen cuatro multiplicaciones reales —cada multiplicación de dos complejos implica cuatro multiplicaciones reales— y dos sumas por cada multiplicación compleja. Para una misma *k*, se necesitan N multiplicaciones complejas y N-1 sumas complejas —como se ha comentado en el párrafo anterior—, lo que implica *4N* multiplicaciones reales y *2N* sumas reales debidas a las multiplicaciones de complejos. Como en el sumatorio hay de *N-1* sumas complejas y cada una implica dos sumas reales —por un lado la suma de las partes reales y por otra la de las partes imaginarias—, hay en total *2(N-1)* sumas reales que junto con las sumas debidas a la multiplicación hace un total de *2N+[2(N-1)] = 4N-2* sumas reales. Ya que la *k* toma *N* valores diferentes, se tiene en total $4N^2$ multiplicaciones reales y *N(4N-2)* sumas reales. Estos datos son aproximados [19] ya que habrá factores que sean uno —lo que realmente no necesitará una operación de multiplicación— y se han considerado las muestras temporales —*x[n]*— como complejas —aunque lo habitual es que sean reales—, sin embargo, estos datos dan una idea de lo elevado que puede llegar a ser el desarrollo mediante el cálculo directo de (4.28) más atrás.

Para simplificar los cálculos se va a explotar las características de periodicidad y simetría de los factores W_N^{nk}. Por simplicidad se va a tomar la *k* para el valor uno [18]:

$$W_N^{n+N} = W_N^n, \quad periodicidad \tag{4.29}$$

$$W_N^{n+\frac{N}{2}} = -W_N^n, \qquad simetria \tag{4.30}$$

Las comprobaciones de estas propiedades se pueden realizar sustituyendo estos factores por las exponenciales que representan y operando:

$$W_N^{n+N} = e^{-j\frac{2\pi}{N}(n+N)} = e^{-j\frac{2\pi}{N}n} \cdot e^{-j\frac{2\pi}{N}N} = e^{-j\frac{2\pi}{N}n} = W_N^n, \quad periodicidad \tag{4.31}$$

$$W_N^{n+\frac{N}{2}} = e^{-j\frac{2\pi}{N}\left(n+\frac{N}{2}\right)} = e^{-j\frac{2\pi}{N}n} \cdot e^{-j\frac{\pi}{2}} = e^{-j\frac{2\pi}{N}n} \cdot (-1) = -W_N^n, \qquad simetria \tag{4.32}$$

Otras propiedades de interés son las siguientes:

$$W_N^2 = e^{-j\frac{2\pi}{N}2} = e^{-j\frac{2\pi}{N/2}} = W_{N/2} \tag{4.33}$$

$$W_N^{Nn} = e^{-j\frac{2\pi}{N}(Nn)} = e^{-j\frac{2\pi N}{N}n} = e^{-j2\pi \cdot n} = e^{-j\frac{2\pi}{N}\cdot 0} = W_N^0 = 1 \tag{4.34}$$

[22] Realmente lo que se maneja en una máquina es una aproximación a los números reales mediante números racionales, aunque, para este análisis, se puede hacer la asunción que se trata de números reales.

El algoritmo parte de un registro en el dominio de la frecuencia, de tamaño *N* muestras, que ha de ser potencia de dos, y la estrategia a es ir descomponiéndolo la secuencia en el dominio de la frecuencia en secuencias de tamaño mitad —una secuencia mitad con los elementos pares de la original, y la otra con los impares—, de forma sucesiva, hasta llegar a secuencias de tamaño dos muestra. Para realizarlo, inicialmente se divide la expresión de la TDF —(4.26) más atrás— en dos sumatorios, el primero de los cuales tendrá la primera mitad de las muestras temporales mientras que la segunda, la mitad restante —la segunda mitad—:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{nk} = \sum_{n=0}^{N/2-1} x[n]\, W_N^{nk} + \sum_{n=N/2}^{N-1} x[n]\, W_N^{nk} \qquad (4.35)$$

Se realiza un cambio de variable en el segundo sumatorio para que el rango del índice sea igual al del primer sumatorio:

$$X[k] = \sum_{n=0}^{N/2-1} x[n]\, W_N^{nk} + \sum_{n=0}^{N/2-1} x\left[n+\frac{N}{2}\right] W_N^{\left(n+\frac{N}{2}\right)k} \qquad (4.36)$$

Se expresa el factor de rotación del segundo sumando como multiplicación de factores:

$$X[k] = \sum_{n=0}^{N/2-1} x[n]\, W_N^{nk} + \sum_{n=0}^{N/2-1} x\left[n+\frac{N}{2}\right] W_N^{nk} \cdot W_N^{\frac{N}{2}k} \qquad (4.37)$$

Ya que el segundo factor de rotación del segundo sumando no depende del índice del sumatorio —*n*—, se saca fuera del mismo:

$$X[k] = \sum_{n=0}^{N/2-1} x[n]\, W_N^{nk} + W_N^{\frac{N}{2}k} \cdot \sum_{n=0}^{N/2-1} x\left[n+\frac{N}{2}\right] W_N^{nk} \qquad (4.38)$$

Aplicando la propiedad de simetría —(4.30) más atrás —, el factor $W_N^{N/2k}$ se puede sustituir por $(-1)^k$:

$$X[k] = \sum_{n=0}^{N/2-1} x[n]\, W_N^{nk} + (-1)^k \cdot \sum_{n=0}^{N/2-1} x\left[n+\frac{N}{2}\right] W_N^{nk} \qquad (4.39)$$

Finalmente se pueden reunir estos dos sumatorios en uno solo:

$$X[k] = \sum_{n=0}^{N/2-1} \left[x[n] + (-1)^k \cdot x\left[n+\frac{N}{2}\right]\right] \cdot W_N^{nk} \qquad (4.40)$$

Con el sumatorio así expresado, se puede proceder al diezmado de la secuencia en el dominio de la frecuencia —*X[k]*— en una secuencia con muestras pares y otras impares. La secuencia con muestras pares es:

$$X[2k] = \sum_{n=0}^{N/2-1} \left[x[n] + x\left[n+\frac{N}{2}\right]\right] \cdot W_N^{n2k} = \sum_{n=0}^{N/2-1} \left[x[n] + x\left[n+\frac{N}{2}\right]\right] \cdot W_{N/2}^{nk}, \quad k = 0,1,\dots,N/2-1 \tag{4.41}$$

En el coeficiente de rotación del sumatorio de la expresión anterior se ha aplicado la propiedad de la expresión (4.33) más atrás. La secuencia con muestras impares es:

$$\begin{aligned} X[2k+1] &= \sum_{n=0}^{N/2-1} \left[x[n] - x\left[n+\frac{N}{2}\right]\right] \cdot W_N^{n(2k+1)} \\ &= \sum_{n=0}^{N/2-1} \left[x[n] - x\left[n+\frac{N}{2}\right]\right] \cdot W_N^{n2k} \cdot W_N^{n} \\ &= \sum_{n=0}^{N/2-1} \left[x[n] - x\left[n+\frac{N}{2}\right]\right] \cdot W_{N/2}^{nk} \cdot W_N^{n} \\ &= \sum_{n=0}^{N/2-1} \left[\left[x[n] - x\left[n+\frac{N}{2}\right]\right] \cdot W_N^{n}\right] \cdot W_{N/2}^{nk}, \qquad k = 0,1,\dots,N/2-1 \end{aligned} \tag{4.42}$$

En esta expresión, inicialmente se ha plasmado el coeficiente de rotación con exponente de una suma como multiplicación de coeficientes. Luego, en uno de ellos se ha aplicado la propiedad (4.33) más atrás, y finalmente se han reordenado convenientemente para que el sumatorio de muestras impares tenga una apariencia similar al de las muestras pares. Hay que señalar que ambas secuencias —secuencia de muestras pares y secuencia de muestras impares— están definidas para N/2 muestras. Ambos sumatorios se pueden redefinir teniendo en cuenta las siguientes secuencias temporales:

$$\begin{aligned} x_1[n] &= x[n] + x\left[n+\frac{N}{2}\right] \\ x_2[n] &= \left[x[n] - x\left[n+\frac{N}{2}\right]\right] \cdot W_N^{n} \end{aligned} \qquad n = 0,1,\dots,\frac{N}{2}-1 \tag{4.43}$$

$$\begin{aligned} X[2k] &= \sum_{n=0}^{N/2-1} \left[x_1[n]\right] \cdot W_{N/2}^{nk} \\ X[2k+1] &= \sum_{n=0}^{N/2-1} \left[x_2[n]\right] \cdot W_{N/2}^{nk} \end{aligned} \qquad k = 0,1,\dots,\frac{N}{2}-1 \tag{4.44}$$

Con esta nueva reorganización —fórmulas (4.44) más atrás — se ha conseguido expresar las muestras pares e impares en el dominio de la frecuencia en función de sendas secuencias temporales que son subgrupos de la secuencia temporal original —expresiones (4.43) más atrás—. En consecuencia, la TDF original de N puntos se puede calcular como dos TDF de N/2 puntos. Observando las expresiones (4.43) más atrás, se aprecia que cada muestra de la secuencia frecuencial — expresiones (4.44) más atrás — se calcula como la suma y resta de la primera mitad de la secuencia temporal original —x[n]— con la segunda mitad de la secuencia temporal original —x[n+N/2]—. Esto deriva en el siguiente esquema:

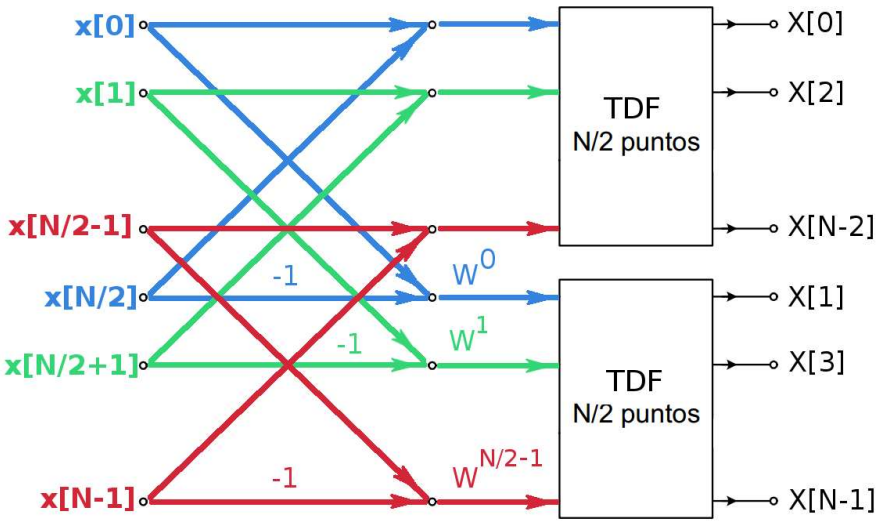


Ilustración 4.39: Esquema general del diezmado en frecuencia

En él se aprecia estas formas de cálculo conocidas como «mariposa», donde en cada dato de la secuencia de salida se tiene en cuenta el dato de la secuencia de entrada, y el dato de esta misma secuencia que está separada en N/2 puntos —(4.43) más atrás—.

Concretando para el caso de 8 puntos, el esquema sería [19]:

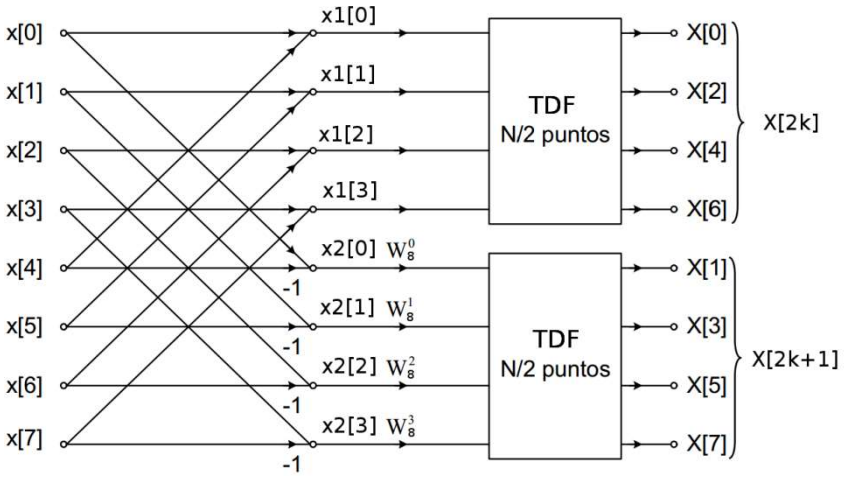


Ilustración 4.40: Esquema del primer diezmado para ocho muestras en la FFT

En este esquema se muestra como una FFT de ocho puntos se puede calcular como dos TDF de N/2 puntos, separando la secuencia de salida en las muestras pares e impares.

Si se procede a un segundo diezmado —de N/4 puntos— de las dos TDF de N/2, se obtiene el siguiente esquema [19]:

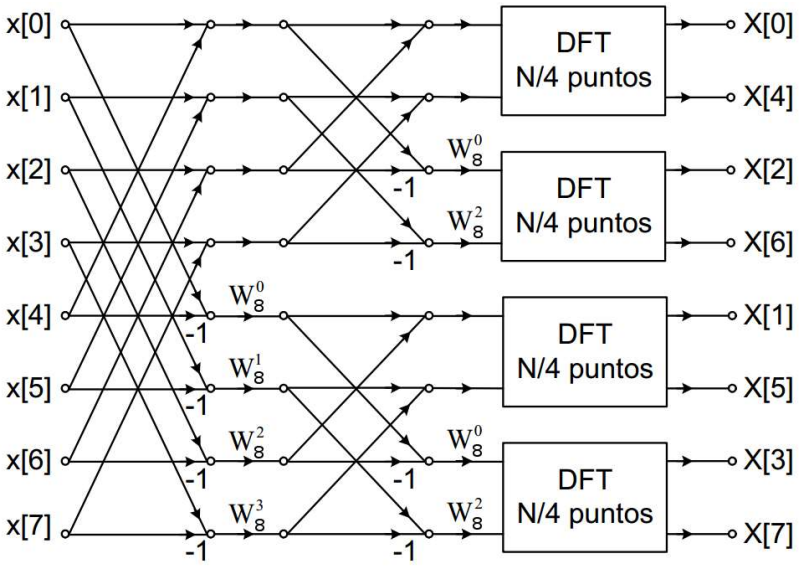


Ilustración 4.41: Esquema del segundo diezmado para ocho muestras en la FFT

Cada dos TDF contiguas de N/4 puntos separa la secuencia de salida en muestras pares e impares respecto de las muestras donde se produce el diezmado. Así, para el caso de las dos TDF superiores de la ilustración anterior, que diezman la TDF superior de N/2 puntos de la Ilustración 4.41—en esta última TDF solo estaban la muestras de salida pares de la TDF completa—, se separa, en la superior las muestras pares —de las pares de la secuencia completa— y en la inferior las impares —de las pares de la secuencia completa—.

Y si finalmente se procede al diezmado de las TDF de N/8 se obtiene:

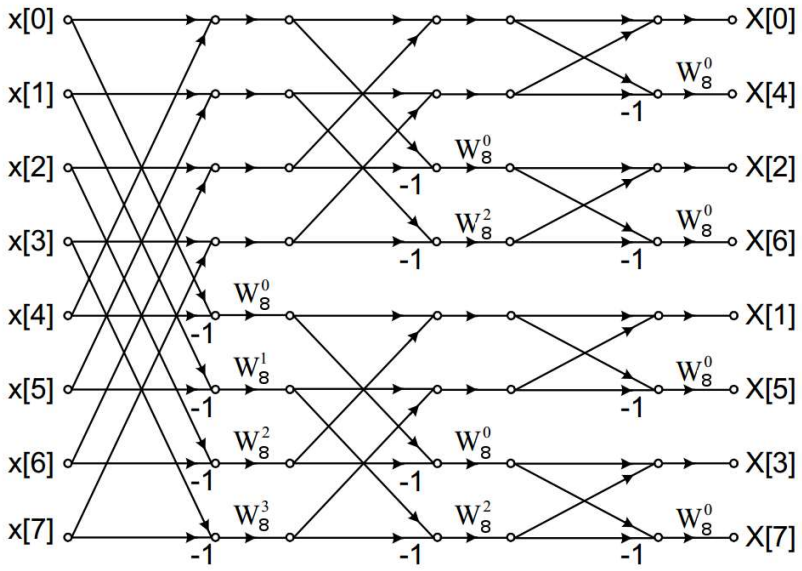


Ilustración 4.42: Esquema del tercer diezmado para ocho muestras en la FFT

Como se aprecia en esta ilustración, y como se desprende de las expresiones ()4.43 más atrás, para el caso de TFDs de dos muestras, se calcula la salida par como la suma de las dos entradas temporales y la salida impar de la TFD de como la diferencia. También se observa que las salidas de la TFD entera no han quedado ordenadas. Pero pueden ser fácilmente ordenadas si se invierte el orden del los bits del índice de salida. Así, por ejemplo, la salida que aparece en sexto lugar es la salida X[3]. Si se recorre la salida con un índice que se incremente de 0 a 7, no se obtiene el orden correcto de la muestra, pero si una vez producido el incremento de este índice, invertimos sus bits, se recorrerá la secuencia de salida en orden correcto. Así si invertimos los bits del número 3 —011—, se obtiene el número 6 —110—, lugar donde se encuentra la muestra de salida X[3]. Esto también se puede conseguir mediante un array asociativo donde cada dato indexado corresponda con el índice de salida. Esta última opción es la que utiliza CMSIS.

En definitiva, este algoritmo de diezmado se repite $\log_2(N)$ veces ya que el tamaño de la secuencia es potencia de dos, con lo que el número total de multiplicaciones complejas es de $(N/2)\log_2(N)$ y el de sumas compleja de (N)log2(N) [12].

4.5.2 CMSIS DSP y la FFT.

CMSIS es el acrónimo en lengua inglesa de «Cortex Microcontroller Software Interface Standar» —interfaz de software estándar de microcontroladores Cortex— y es una capa de abstracción de hardware que independiza el modo de acceso a los periféricos del micro, respecto del fabricante concreto que los implementa. Así se tienen periféricos comunes para todos los fabricantes, como puedan ser el reloj del sistema operativo, SysTick, o el controlador de vectores de interrupción anidadas —NVIC—, aunque este último con ligeras diferencias entre un fabricante y otro, donde CMSIS permite su uso de forma independiente del hardware. Para el caso de los periféricos propios de cada fabricante, CMSIS da unas líneas guía para poder ampliar la librería. Así, por ejemplo, está la *librería estándar de drivers para periféricos* de micros *ST*, donde se extiende la librería CMSIS para los periféricos concretos de este fabricante —como puedan ser GPIO, ADC, DAC, DMA, DMA2D...— o la más reciente librería STM32Cube, aunque esta última no sigue las recomendaciones CMSIS en la secuencia de inicialización del reloj sistema.

CMSIS además de dar acceso a periféricos, ofrece varia funcionalidad que ayuda al programador. En el caso tratado aquí se ha usado la parte de esta librería para el procesamiento digital de señales: CMSIS-DSP, y concretamente aquella dedicada a la transformada rápida de Fourier. En CMSIS-DSP se proveen alrededor de 60 funciones repartidas entre las que manejan tipos en coma fija —para Cortex M0, M3 y M4— y las que manejan coma flotante de precisión simple —Cortex M4 con periférico FPU (unidad de coma flotante), y en las versiones más recientes, soporte para Cortex M7—. Algunas de estas funciones han quedado obsoletas y han sido reemplazadas por otras nuevas más eficientes —aunque las obsoletas se mantienen, por el momento, por motivos de compatibilidad—.

En cualquier caso, para poder realizar la FFT se requieren dos pasos: un proceso de inicialización, donde se definen los valores iniciales de las estructuras que utiliza el desarrollo, y el cálculo de la FFT en sí mismo.

CMSIS divide este cálculo en aquellas secuencias de entrada que son complejas —se denomina a la transformada como cfft— y aquellas que son reales —la transformada es llamada como rfft—. Para la entrada de datos complejos el formato intercala la parte real y la imaginaria en el array de entrada. Por ello, este array debe ser de tamaño doble al tamaño de datos de la FFT que se quiera realizar. Así, si se quiere realizar una FFT de 64 datos complejos de entrada, se necesitará un array de 128 datos de tipo flotante de 32 bits, ya que habrá 64 datos pertenecientes a la parte real y otros 64 a la parte imaginaria. La salida del cálculo de la FFT se realizará en el mismo array de entrada con el mismo formato intercalado —parte real del primer dato, parte imaginaria del

primer dato, parte real del segundo dato, parte imaginaria del segundo dato …—. El algoritmo que utiliza el cálculo para secuencias de entrada compleja es algo más complicado que el presentado en el apartado 4.5.1, aunque conceptualmente se trata igualmente de un diezmado en frecuencia. El presentado en el apartado 4.5.1 es el conocido como algoritmo *radix-2*, en cambio, el utilizado por CMSIS es una combinación de *radix-2*, *radix-4* y *radix-8*, aunque se manejan los mismos conceptos de factores de rotación e inversión de bits. Los tamaños de datos permitidos —tanto en la FFT de entrada compleja como de entrada real— son las potencias de dos que van desde 16 datos a 4096 datos. En este proyecto se ha utilizado una entrada real de 2048 datos.

Para la inicialización de la FFT con entrada compleja se utiliza una estructura donde se especifica la tabla a utilizar de los factores de rotación e inversión de bits. Esta estructura está definida en el archivo de cabecera *arm_math.h* y se ha de incluir en todos aquellos archivos fuente donde se utilice este cálculo. Esta estructura se muestra a continuación [20]:

```
1  typedef struct
2    {
3      uint16_t fftLen;                    /**< length of the FFT. */
4      const float32_t *pTwiddle;          /**< points to the Twiddle factor table. */
5      const uint16_t *pBitRevTable;       /**< points to the bit reversal table. */
6      uint16_t bitRevLength;              /**< bit reversal table length. */
7    } arm_cfft_instance_f32;
```

Código 4.98: Estructura de inicialización de la CFFT en CMSIS-DSP

El primer miembro de esta estructura —*fftLen*— es el tamaño del registro de entrada —en número de datos complejos—; el segundo, es el puntero a la tabla de factores de rotación; el tercero es el puntero a la tabla de inversión de bits, y el último, el tamaño de esta última tabla. Todas estas tablas están definidas en el archivo *arm_common_tables.c*, y tiene la siguiente estructura [20]:

```
1  const uint16_t armBitRevTable[1024] = {
2     0x400, 0x200, 0x600, 0x100, 0x500, 0x300, 0x700, 0x80, 0x480, 0x280,
3  . . .}
4  const float32_t twiddleCoef_16[32] = {
5      1.000000000f,  0.000000000f,
6  . . .}
7  . . .
8  const float32_t twiddleCoef_4096[8192] = {
9      1.000000000f,  0.000000000f,
10 . . .}
11 . . .
12 const uint16_t armBitRevIndexTable16[ARMBITREVINDEXTABLE__16_TABLE_LENGTH] =
13 {
14    //8x2, size 20
15    8,64, 24,72, 16,64, 40,80, 32,64, 56,88, 48,72, 88,104, 72,96, 104,112
16 };
17 . . .
18 const  uint16_t  armBitRevIndexTable_fixed_4096[ARMBITREVINDEXTABLE_FIXED_4096_TABLE_LENGTH]
19 =
20 {
21     //radix 4, size 4032
22     8,16384, 16,8192, 24,24576, 32,4096, 40,20480, 48,12288, 56,28672, 64,2048,
23 . . .}
24 . . .
```

Código 4.99: Definición de tablas de factores de rotación e inversión de bits en CMSIS-DSP

En la primera línea aparece la parte inicial de la tabla de inversión de bits —armBitRevTable— que utilizan las funciones obsoletas de inicialización de la cfft como son *arm_cfft_radix2_init_f32* y *arm_cfft_radix4_init_f32c* —además de las versiones equivalentes para datos de coma fija—. La segunda y tercera tabla —líneas 4 y 8— son la parte inicial de las tablas de coeficientes de rotación para los tamaños 16 y 4096 datos respectivamente —las tablas para otros tamaños de datos, que son igualmente potencias de dos y que están entre las dos mostradas, se han omitido por simplicidad del código mostrado—. En las líneas 12 y 18 se muestran la parte inicial de las tablas de inversión de bits para los casos de 16 y 4096 datos. Igualmente, se han omitido las de inversión de bits de los tamaños comprendidos entre las mostradas. Estas tablas, así como las constantes que fijan sus tamaños están

definidas de forma externa en el archivo de cabecera *arm_common_tables.h* para que puedan ser accedidas desde cualquier archivo que incluya esta cabecera. Teniendo en cuenta estas tablas, ya se puede rellenar la estructura de tipo *arm_cfft_instance_f32* —Código 4.98—, pero para evitar errores —como que en la misma estructura se especifique una tabla para factores de rotación para un tamaño de datos y una tabla de inversión de bits para otro— CMSIS-DSP provee un archivo de cabecera donde están definidas todas las posibles estructuras de tipo *arm_cfft_instance_f32*, esta cabecera es *arm_const_struct.h*. Por ejemplo, la estructura para una FFT de 2048 datos esta especificada como [20]:

```
1  #include "arm_math.h"
2  #include "arm_common_tables.h"
3
4  . . .
5  extern const arm_cfft_instance_f32 arm_cfft_sR_f32_len2048;
6  . . .
```

Código 4.100: Estructura de inicialización de la cfft para 2048 datos en CMSIS-DSP

Esta definición externa corresponde a la siguiente definición en el archivo fuente *arm_const_struct.c* [20]:

```
1  #include "arm_const_structs.h"
2  . . .
3  const arm_cfft_instance_f32 arm_cfft_sR_f32_len2048 = {
4     2048, twiddleCoef_2048, armBitRevIndexTable2048, ARMBITREVINDEXTABLE2048_TABLE_LENGTH
5  };
6  . . .
```

Código 4.101: Definición de la estructura de inicialización de la cfft para 2048 datos en CMSIS-DSP

De esta forma, la inicialización de la cfft es la simple mención de la estructura para el tamaño de la FFT deseada —mirar *arm_const_struct.h* para ver las otras estructuras disponibles—.

Para calcular la FFT de entrada compleja, se pasa como argumento, a la función *arm_cfft_f32* —función encargada del cálculo de la TDF para entrada compleja—, la estructura de inicialización antes comentada, la dirección del array de entrada/salida de la secuencia; si se va a realizar la transformada inversa —para cálculo inverso este argumento ha de ser uno o cero para el caso contrario (transformada directa) — y si en la salida se va a realizar la inversión de bits —valor uno y en consecuencia, en orden correcto— o no usar inversión de bits —valor cero y salida en el orden inverso de bits para los índices del array—. Por ejemplo, teniendo un array de 4096 datos en coma flotante de 32 bits —en formato con partes reales intercaladas con las imaginarias y por tanto, la longitud de los datos complejos es de 2048—, para calcular la FFT directa con salida en orden correcto de datos se utilizaría un código similar al siguiente:

```
1  #include "arm_math.h"
2  . . .
3  float registro[4096];
4  . . .
5  //código de relleno de la secuencia compleja de entrada en «registro»
6  arm_cfft_f32(arm_cfft_sR_f32_len2048, registro, 0, 1);
7  . . .
```

Código 4.102: Cálculo de la FFT de entrada compleja con CMSIS-DSP

En la línea 6 se realiza el cálculo de la FFT de entrada compleja de longitud 2048 —primer argumento, donde además están implícitamente especificadas las tablas de factores de rotación e inversión de bits para este tamaño—, la secuencia de entrada está en el array *registro* —segundo argumento—, se realiza la transformada directa —tercer argumento— y la salida estará en orden correcto —cuarto argumento—. La salida de la FFT

queda almacenada en el array *registro*, por lo que se ha sobrescrito el contenido de la secuencia compleja de entrada.

Para el caso de cálculo de la FFT de entrada real —la utilizada en este proyecto— CMSIS-DSP usa la cfft y se aprovecha la propiedad de simetría en el espectro en una secuencia real —la segunda mitad del espectro es la compleja conjugada de la primera— realizando la transformada real de forma más rápida que la equivalente de entrada compleja. Para ello, sigue este esquema en la transformada directa, sacado de la ayuda de CMSIS:

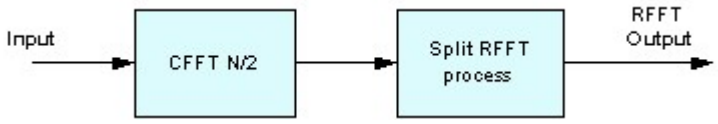


Ilustración 4.43: Etapas del cálculo de la FFT real en CMSIS-DSP

Comprende una primera etapa de N/2 de longitud, donde calcula la mitad de la transformada compleja —las partes imaginarias de la secuencia real de entrada se toman como cero— y luego, en la segunda etapa, se reconstruye la otra mitad del espectro. El proceso del cálculo de la FFT inversa real es similar, pero no se va a comentar ya que en este proyecto no ha sido usado.

La rfft —la FFT de entrada real—utiliza la siguiente estructura de inicialización [20]:

```
1  typedef struct
2    {
3      arm_cfft_instance_f32 Sint;  /**< Internal CFFT structure. */
4      uint16_t fftLenRFFT;         /**< length of the real sequence */
5      float32_t * pTwiddleRFFT;    /**< Twiddle factors real stage  */
6    } arm_rfft_fast_instance_f32 ;
```

Código 4.103: Estructura de inicialización de la RFFT en CMSIS-DSP

Como primer miembro tiene la estructura de inicialización de la cfft —línea 3—, ya que en el proceso de cálculo de la rtff está involucrado un desarrollo cfft. El segundo miembro es el tamaño del conjunto de datos reales de entrada —línea 4—. Dentro del primer miembro de esta estructura y que, como se ha dicho, corresponde con la estructura de inicialización de la FFT compleja, hay, a su vez, un miembro que también alberga el tamaño de los datos de entrada, pero en este segundo caso, este tamaño es mitad del especificado en la línea 4 del código anterior. El último miembro —línea 5— corresponde al puntero a la tabla de factores de rotación que utiliza exclusivamente la etapa de reconstrucción de la segunda mitad del espectro —Split RFFT process de la Ilustración 4.43—.

El proceso de inicialización del la rfft no es tan simple como el de la cfft, donde se usaban estructuras ya creadas para los distintos casos. En la rfft se utiliza la función de inicialización *arm_rfft_fast_init_f32()* que recibe como argumentos el puntero a la estructura mostrada en el código anterior y el tamaño de la longitud de los datos de la secuencia real. A partir del valor de este segundo argumento —el tamaño de la secuencia real— rellena los datos de la estructura pasada como primer argumento, es decir, establece el tamaño de la secuencia compleja a la mitad del tamaño de la real; establece la tabla de factores de rotación e inversión de bits para la etapa cfft y establece la tabla de factores de rotación para etapa de reconstrucción. Una vez inicializada la estructura, se procede al cálculo de la FFT real mediante la función *arm_rfft_fast_f32()* que espera como primer argumento la estructura inicializada, como segundo , el puntero al array con la secuencia real de entrada, el puntero al array de salida y finalmente, si se trata de la transformada directa o la inversa. En este sentido, tiene la

gran diferencia respecto de la cfft, que necesita un array de entrada y otro de salida. Un ejemplo de código de utilización de la rfft es el siguiente:

```
1  #include "arm_math.h"
2  #define N_DATOS_REALES 2048
3  . . .
4  float registro entrada[N DATOS REALES];   //albergará la secuencia real de entrada
5  float registro salida[N DATOS REALES*2];  // albergará la fft compleja de salida
6  arm_rfft_fast_instance_f32 EstructuraRFFT;
7  . . .
8  //código de relleno de la secuencia compleja de entrada en «registro_entrada»
9
10 arm_rfft_fast_init_f32(&EstructuraRFF, N_DATOS_REALES); // inicializa la estructura
11 /* se clacula la fft */
12 arm_rfft_fast_f32(&EstructuraRFF,  registro_entrada, registro_salida, 0);
13 . . .
14 /* se clacula la magnitud de la fft */
15 arm_cmplx_mag_f32(registro_salida, registro_entrada, N_DATOS_REALES);
```

Código 4.104: Cálculo de la FFT de entrada real con CMSIS-DSP

Primeramente se ha de disponer de los arrays de entrada y salida —líneas 4 y 5 respectivamente—, teniendo en cuenta que, aunque la entrada sea real, la salida será compleja y necesitará el doble de tamaño —ya que tendrá parte real y parte imaginaria—. También se debe disponer de la variable de estructura de inicialización que espera el cálculo de la rfft, aunque inicialmente los miembros de esta estructura tengan valores no útiles —estructura con los valores sin inicializar—.. Luego, en la línea 10, se procede a inicializar esta estructura con el solo dato del tamaño de la secuencia de entrada. Una vez hecho esto, ya se puede calcular la FFT, quedando el resultado en el array de salida —línea 12—. Finalmente, como habitualmente en lo que se está interesado es en el cálculo de la magnitud de la FFT, se procede a ello —línea 15—, aprovechando el array de entrada de la secuencia temporal, como array de salida de la magnitud de la FFT. Esta misma metodología —el uso de los dos registros y el cálculo de la magnitud— será la usada en este proyecto cuando se calcule y presente la FFT en pantalla.

4.5.3 Manejo del registro FFT y ventanas de ponderación temporales.

Una vez realizada la captura temporal del registro, es necesario interpretar los datos contenidos en él de forma correcta. Esto implica conocer qué es lo que realmente contiene y el significado que tienen sus valores —como qué nivel representan y a qué frecuencia o tiempo corresponden—. El siguiente esquema resume lo que contiene el registro en cada momento:

Ilustración 4.44: Secuencia de datos en el registro

Inicialmente se capturan 2048 datos enteros mediante el periférico interno ADC del micro en configuración triple entrelazada —se explicará esta configuración en el apartado 4.5.5—. Estos datos son de tipo entero sin signo de ocho bits en el rango de entrada de 0 a 3 voltios y son capturados de forma alternada en dos arrays llamados *registro1* y *registro2*—técnica de doble buffer— para que, mientras uno se use para la captura, el otro se use para el procesamiento de datos. Estos arrays, así como otros dedicados al procesamiento —e implícitamente contenidos en la ilustración anterior— están declarados en el archivo *DatosEntreTareas.cpp*, un archivo fuente que contiene datos que se comparten entre la tarea gráfica y la tarea de la aplicación y que será mostrado en el apartado 4.5.10. A continuación, se convierten a datos en coma flotante, ya que en las funciones elegidas para el cálculo de la FFT se utiliza este tipo de datos. Seguidamente se procede al enventanado temporal, que consiste en multiplicar la captura por una función de ponderación. Esto hace que se atenúen los datos iniciales y finales del registro para minimizar la discontinuidad —ya que la transformada discreta es periódica tanto en el dominio del tiempo como en el de la frecuencia [20]—. Después, se realiza el cálculo de la FFT que da como resultado una secuencia compleja de misma longitud que la de entrada, es decir, ya que la entrada es de 2048 —datos reales, o vistos de otro modo, 2048 datos complejos con parte imaginaria cero—, la salida es igualmente de 2048 datos complejos —que equivalen, en cantidad de memoria a 4096 datos de coma flotante—. Para la representación del espectro, lo interesante es la magnitud de la transformada, por ello, el siguiente paso que se realiza es el cálculo del valor absoluto de los datos, dando como resultado un registro de 2048 datos reales. Finalmente los datos son vueltos a convertir en enteros y solo se muestra la primera mitad del registro, ya que la magnitud del espectro obtenido es simétrica respecto de la frecuencia cero —espectro positivo y negativo—. Esto último está íntimamente relacionado con el teorema de Nyquist. Estos son los pasos más

importantes que se llevan a cabo sobre el registro, pero hay otros —como el cálculo del logaritmo de los datos o el escalado gráfico— que se explicaran con más detenimiento en la sección 4.5.8—.

Una vez obtenido la magnitud del espectro, hay que determinar a qué frecuencias corresponde cada dato. La separación de frecuencia entre dato y dato es conocida como delta de frecuencia y está relacionada con la frecuencia o periodo de muestreo mediante la siguiente expresión [14]:

$$\Delta F = \frac{F_M}{N} \tag{4.45}$$

Donde $\boldsymbol{\Delta F}$ es el incremento de frecuencia de un dato a otro del espectro —resolución en frecuencia—, $\boldsymbol{F_M}$, la frecuencia de muestreo y N, la cantidad de datos del espectro —en este caso 2048—. La máxima frecuencia que se puede representar es la mitad de la frecuencia de muestreo —teorema de Nyquist—, que correspondería con el dato de índice 1024 —***N/2***—. El siguiente esquema, muestra la relación entre el registro en el dominio del tiempo y el registro en el dominio de frecuencia:

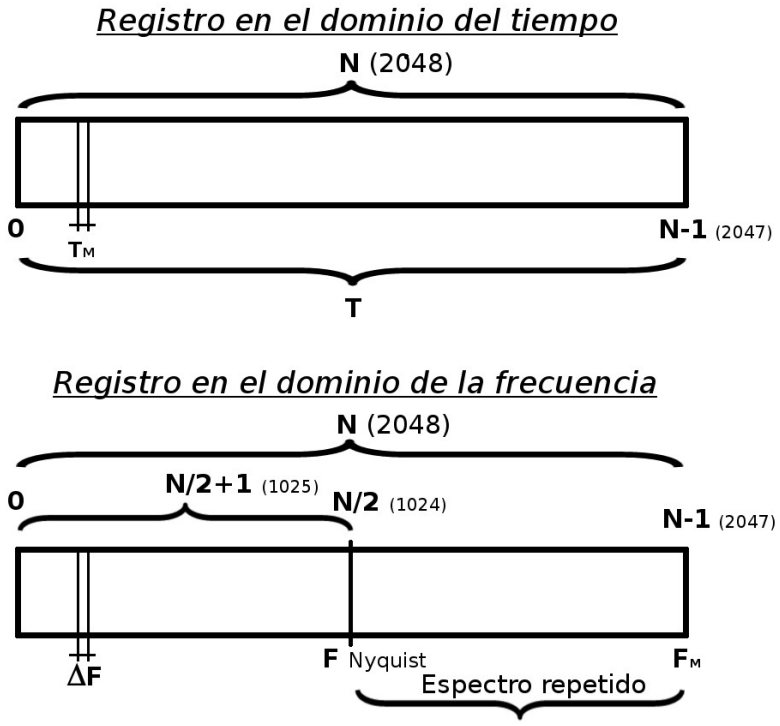


Ilustración 4.45 Relación entre el registro temporal y el frecuencial —magnitud—

En la parte superior de la ilustración se muestra, esquemáticamente, el registro en el dominio del tiempo. Cada muestra, en este registro, estará capturada en múltiplos del periodo de muestreo, o lo que es lo mismo, cada muestra está separada de la siguiente en un tiempo igual al periodo de muestreo —representado por T_M—, con lo que este intervalo de tiempo es el mínimo medible —resolución temporal—. Cuando se calcula la FFT se genera el registro que aparece en la parte inferior de la ilustración. Se obtienen igualmente 2048 datos. La frecuencia de Nyquist aparece en el dato de índice ***N/2*** —índice 1024—. Tanto el dato de índice cero como el de índice 1024, son reales en la transformada [12], y CMSIS-DSP los empaqueta juntos en el primer par cartesiano complejo, poniendo el dato de índice cero en la parte real del par y en la parte imaginaria el dato de índice ***N/2***, pero hay que tener presente que este par cartesiano no representa un número complejo, es solo una forma eficiente de empaquetar los datos. Debido a que el espectro aparece repetido en magnitud, ya que en la

transformada de una secuencia real, la parte negativa del espectro es el complejo conjugado de la parte positiva, solo se utiliza la mitad del registro, con la posibilidad de utilizar también el dato de la posición ***N/2*** que representa la frecuencia de Nyquist. En el caso de este proyecto, se ha utilizado hasta el dato con índice 1023 —en total 1024 datos—, quedando la última frecuencia a representar en la magnitud del espectro como [16]:

$$F_{MAX} = \frac{F_M}{2} - \frac{F_M}{N} = \frac{F_M}{2} - \Delta F \qquad (4.46)$$

Cada dato en el dominio de la frecuencia —magnitud— del registro, es realmente un contenedor —el término anglosajón utilizado es *bin*— que representa la densidad espectral para la frecuencia donde está centrado [12]. Estas frecuencias de centrado de cada contenedor —*bin*—, se calculan como $\boldsymbol{\Delta F \times k}$, siendo $\boldsymbol{\Delta F}$ la resolución de frecuencia y $\boldsymbol{k}$, el número de orden del dato en el registro, adquiriendo los valores comprendidos entre 0 a N/2 [16]. El ancho de estos contenedores es la inversa del ancho de la longitud del número de datos del registro. Ya que el ancho del registro de la magnitud es N/2, el ancho de cada contenedor es 2/N [12]. Como cada contenedor representa la densidad espectral, su área es la amplitud de la componente para la frecuencia $\boldsymbol{k}$ concreta. Existen dos datos —dos contenedores— que tienen una situación especial: los datos de los extremos del registro de la magnitud. Estos dos contenedores extremos son los correspondientes a los índices cero y N/2. Debido a su situación, sus anchos no son de 2/N sino que son la mitad, es decir, 1/N, aunque siguen representado la densidad espectral para la frecuencia donde están localizados y por tanto, están magnificados por un factor de dos. Tienen la particularidad, además de ser datos reales. Para comprender mejor estos conceptos, se presenta a continuación la magnitud del espectro de una posible captura de 32 datos temporales [12]:

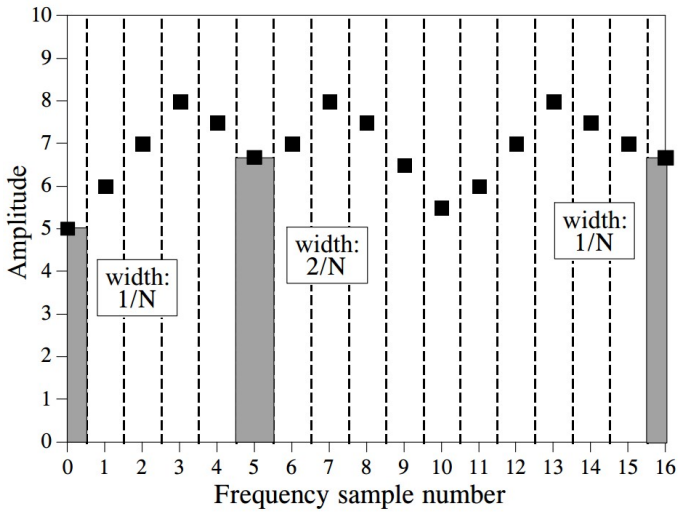


Ilustración 4.46: Ejemplo de magnitud de frecuencia de 32 datos temporales

En esta ilustración se muestra cómo el cálculo de la magnitud de la FFT sobre 32 datos temporales —N datos—, genera 17 datos —N/2+1 datos—. Cada dato de frecuencia —cuadrados negros— está centrado sobre su contenedor —líneas discontinuas verticales—, a excepción de los datos extremos, donde su contenedor es la mitad de ancho que el resto y «la muestra de frecuencia» está en un extremo del mismo, tal y como se muestra en la ilustración para los datos de índice 0 y 16. El área de cada contenedor representa la amplitud de la componente frecuencial sobre la que está situado —aunque en la ilustración aparece en el eje vertical la designación «Amplitude», se refiere a altura—. Por tanto, para calcular la amplitud de cierta componente de

frecuencia, basta con multiplicar la base de cada contenedor por su altura. Por ejemplo, para el caso de la muestra de frecuencia de índice 5, que tiene de altura alrededor de 6,5 y de base 1/16 —2/N = 2/32 = 1/16—,

la amplitud es de 0,40625 —6,5 x 1/16—. En cambio, para un contenedor extremo, como por ejemplo la muestra cero, la amplitud sería de 0,15625 —5 x 1/32—.

Se ha presentado la corrección de los datos en la magnitud del espectro, pero la misma corrección se utiliza sobre el espectro complejo donde cada parte real e imaginaria se multiplica por el factor 2/N, si se trata de una muestra intermedia, o por 1/N, si se trata de una muestra extrema [12]:

$$Re\bar{X}[k] = \frac{ReX[k]}{N/2} \tag{4.47}$$

$$Im\bar{X}[k] = \frac{ImX[k]}{N/2} \tag{4.48}$$

$$Re\bar{X}[0] = \frac{ReX[0]}{N} \tag{4.49}$$

$$Re\bar{X}[N/2] = \frac{ReX[N/2]}{N} \tag{4.50}$$

Los términos $Re\bar{X}$ e Im^{-} hacen referencia a los datos reales e imaginarios, respectivamente, que han sido corregidos para representar la amplitud de las componentes frecuenciales, en oposición a ReX e ImX, que hacen referencia a la densidad espectral de cada una de ellas. Cómo se aprecia en las dos últimas expresiones, correspondientes a las muestras cero y N/2, solo se tiene componente real. Cuando se genera el espectro de frecuencia mediante la TDF, se consigue un espectro discreto equivalente al espectro continuo muestreado cada $\boldsymbol{\Delta F}$ [19]. Esto se aprecia en la expresión real de la TDF:

$$ReX[k] = \sum_{n=0}^{N-1} x[n] cos\left(\frac{2\pi kn}{N}\right)$$

$$k = 0,1,\dots,\frac{N}{2} \tag{4.51}$$

$$ImX[k] = -\sum_{n=0}^{N-1} x[n] sen\left(\frac{2\pi kn}{N}\right)$$

Tanto para la parte real como para la parte imaginaria del espectro, solo se consideran ciertas frecuencias —$2\pi kn/N$— donde las componentes del espectro son determinadas mediante las proyecciones en un conjunto de bases ortogonales representadas por el coseno —coordenada horizontal o real— y el seno —coordenada vertical o imaginaria— distribuidas equidistantes a lo largo del espectro y separadas por $\boldsymbol{\Delta F}$ [13]. Sin embargo, ya que el espectro realmente es continuo, hay frecuencias del mismo que no «caen» exactamente en estas bases. Estas frecuencias son aquellas para las que el valor de la k no es entero —pero la k solo toma valores enteros desde 0 a N/2-1—. Estas otras frecuencias «no muestreadas», no se proyectan en estas bases, pero influyen en todos los contenedores del espectro [13], especialmente a los adyacentes. Esto hace que para una frecuencia «muestreada», se cree a ambos lados unas colas que se extienden por todo el espectro. Este fenómeno se

conoce como fuga espectral —spectral leakage [13]—. Como se acaba de mencionar esto ocurre para frecuencias donde el valor de k no sea entero, lo que implica, en el registro temporal, que no se haya capturado un número de ciclos exactos de la señal. Es posible que la señal que se quiera analizar no sea periódica, pero debido al carácter periódico de la TDF, el registro en el dominio del tiempo se va a tratar como tiempo periódico, de tal forma que el dato que se espera a continuación del último del registro, sea el primer dato, haciendo del registro un registro circular, es lo que se conoce como extensión periódica [13]. La siguiente ilustración muestra esta situación:

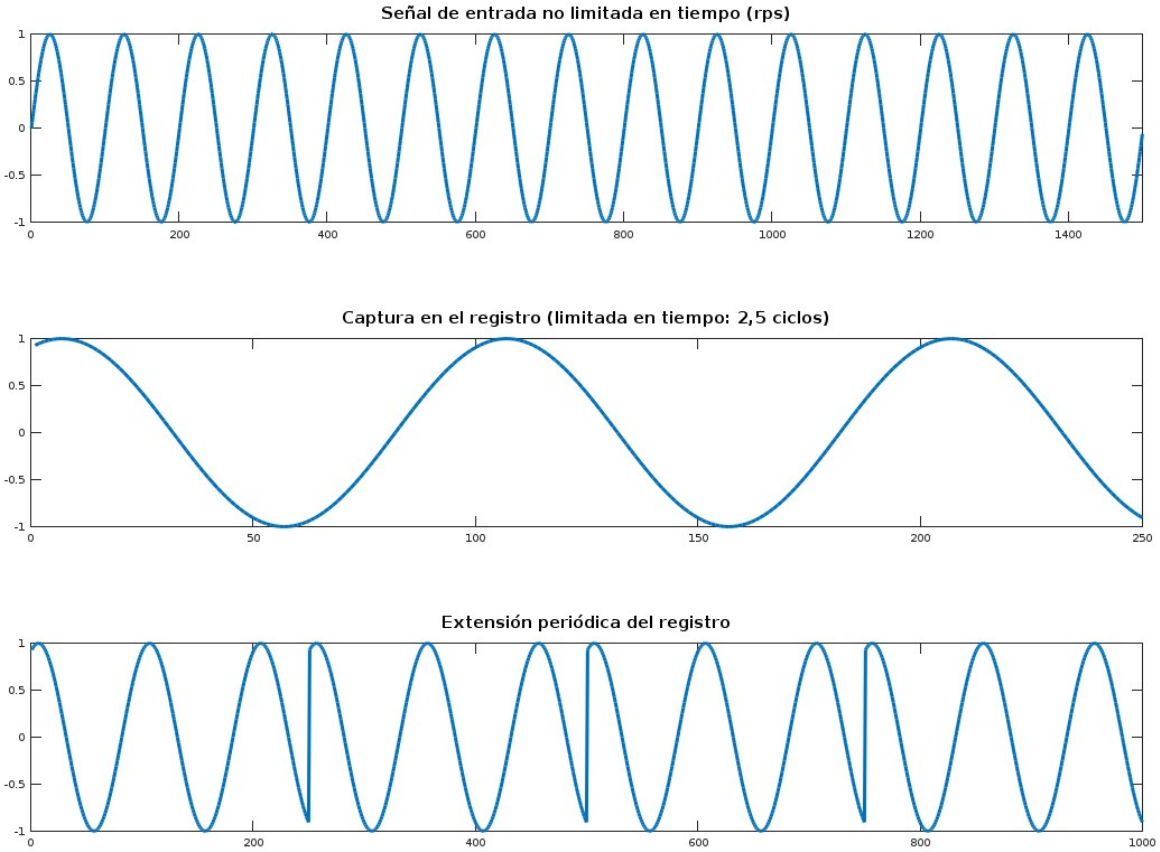


Ilustración 4.47: Extensión periódica del registro temporal

En la gráfica superior de la ilustración anterior, se muestra la señal de entrada al ADC que se asume no limitada en el tiempo —aunque en la gráfica solo se muestran quince ciclos—. Esta gráfica, al igual que las restantes de esta ilustración, muestran los ejes de abscisas con unidades arbitrarias. La gráfica intermedia es la representación del contenido del registro para una captura de la señal de entrada, constituida por dos ciclos y medio de la señal real. En la gráfica inferior está la extensión periódica del registro —donde solo se ha repetido el contenido del registro tres veces, aunque, matemáticamente esta repetición es ilimitada—. El problema que se observa de esta extensión periódica, es que el resultado no corresponde a la señal de entrada. Como se aprecia en la última gráfica, aparecen discontinuidades debidas a la captura finita en tiempo que provoca que la pendiente del último dato de la captura no corresponde con la pendiente del primero, obteniéndose un espectro que no es reflejo del de la señal de entrada —fuga espectral—. La solución es hacer estas pendientes iguales, y una forma sencilla de conseguirlo es tratar que la pendiente de ambos datos sea cero [13]. Ello se consigue mediante una función de ponderación, que multiplicada por el registro, atenúe los datos de los extremos, manteniendo sin atenuación los datos centrales. Esta función es conocida como ventana temporal. En la siguiente figura se muestra el efecto de ponderar los datos capturados en el registro temporal:

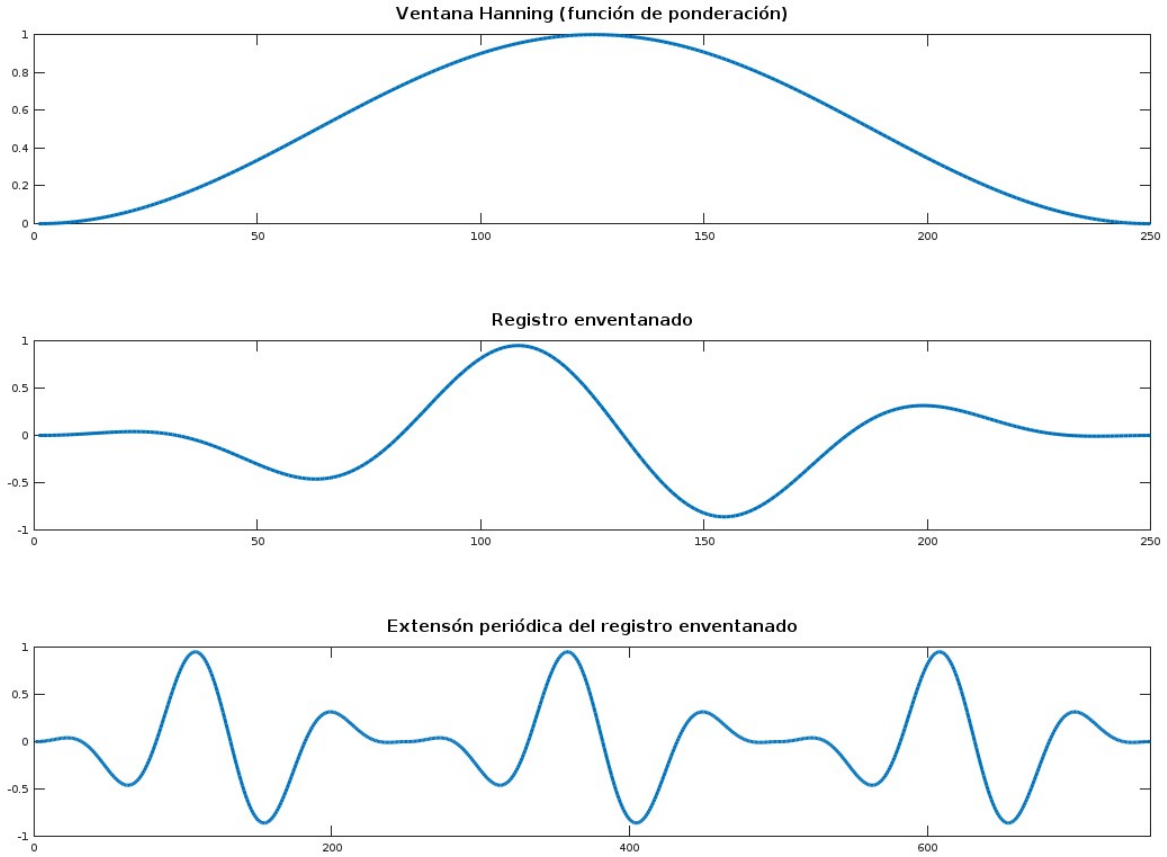


Ilustración 4.48: Ponderación del registro temporal y extensión periódica

En la gráfica superior se muestra la función de ponderación utilizada —se trata de la ventana «Hanning», una de las posibles ventanas de ponderación a utilizar; más adelante se expondrán las ventanas concretas implementadas en este proyecto—. El resultado de la ponderación de los datos del registro —multiplicación de los datos del registro por la función de ventana— se muestra en la gráfica intermedia. Mediante este proceso, se ha conseguido convertir la pendiente de los datos extremos a cero o cercana a cero, lo que permite la extensión periódica del registro sin discontinuidades. Esta extensión periódica es mostrada en la gráfica inferior. Aunque la señal capturada no sea periódica, debido al carácter de la TDF, es necesario que la extensión del registro sí lo sea, por ello, en cualquier caso, es necesario el uso de una función de ventana. La no utilización de una función concreta de ponderación —tal y como se mostraba en la ilustración precedente—, es equivalente a usar una ventana que no atenúa ningún dato, o lo que es lo mismo, que multiplica todos sus datos por la unidad, lo que constituye la ventana conocida como rectangular o uniforme. En las ventanas distintas a la rectangular, debido a la atenuación que originan en los datos extremos del registro temporal, se introduce una pérdida de energía que produce un error en la magnitud del espectro [14], lo que necesita la aplicación de una corrección de nivel como se mostrará más adelante.

Cada ventana de ponderación tiene unas características concretas en el dominio de la frecuencia, como puedan ser la amplitud del lóbulo principal, la anchura del mismo, la amplitud de los lóbulos secundarios o la velocidad de caída de las colas laterales —las características de amplitud están expresadas en decibelios respecto del pico máximo de la magnitud del espectro de la ventana—. Pero siempre se busca mantener un compromiso entre resolución —el ancho del lóbulo principal— y la fuga espectral —la amplitud de las colas— [12]. De tal forma que si se pretende tener buena resolución en frecuencia —lóbulo principal estrecho— se obtienen grandes

colas laterales y viceversa. Así, para este proyecto, se han implementado las ventanas Rectangular, Hanning, Hamming, Blackman-Harris y Barlett —o triangular—[23]:

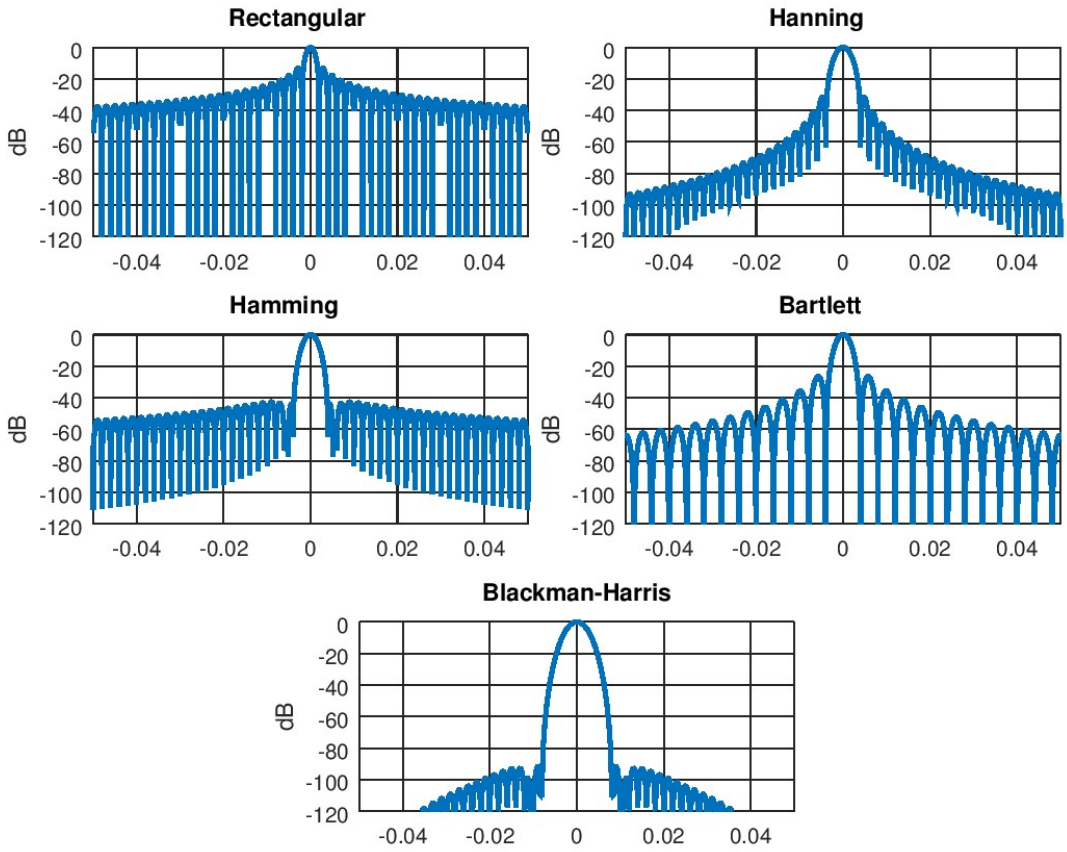


Ilustración 4.49: Respuesta en frecuencia de las ventanas temporales utilizadas

La Rectangular es la que provee mejor resolución en frecuencia pero la de mayor amplitud en las colas laterales; la Hanning y Blackman-Harris mantienen el mejor compromiso entre resolución y amplitud de las colas y la Hamming y Barlett tienen mejor resolución en frecuencia —sin llegar al caso de la rectangular— con grandes colas laterales —sobre todo la triangular—.

Las ventanas Hanning, Hamming y Blackman-Harris, se implementan siguiendo la siguiente expresión [13] [14]:

$$w[n] = \sum_{i=0}^{m} \left((-1)^i a_i \cdot cos\left(\frac{2\pi \cdot i \cdot n}{N}\right)\right), \qquad n = 0,1,\dots,N-1 \qquad (4.52)$$

Donde **w[n]**, representa la secuencia de la ventana; **n**, es el índice de cada dato de la ventana; **N**, el número de datos totales de la misma; **m**, el número de coeficientes menos uno —orden de la ventana [14]— utilizados en la ventana concreta; e **i**, el índice de cada uno de estos coeficientes.

[23] El eje horizontal es la relación entre la frecuencia y la máxima medible —frecuencia de Nyquist—. Las gráficas han sido creadas a partir de un registro de 10000 datos con una anchura de ventana de 500 puntos.

Los coeficientes para cada una de estas ventanas, se resume en la siguiente tabla [13] [14]:

	Hanning	Hamming	Blackman-Harris
m	1	1	3
a_0	0,500000	0,543478	0,35875
a_1	0,500000	0,456522	0,48829
a_2	-	-	0,14128
a_3	-	-	0.01168

Tabla 19: Coeficientes de las ventanas Hanning, Hamming y Blackman-Harris

Para el caso de la ventana triangular, su expresión es [13]:

$$w[n] = \begin{cases} \dfrac{n}{N/2} = \dfrac{2n}{N}, & n = 0,1,\dots,\dfrac{N}{2} \\ w[N-n] = \dfrac{N-n}{N/2} = 2 - \dfrac{2n}{N}, & n = \dfrac{N}{2},\dots,N-1 \end{cases} \tag{4.53}$$

En este caso, la ventana consiste en una pendiente positiva constante hasta el índice N/2 y de ahí hasta el índice N-1, en una pendiente constante negativa.

Como ya ha sido comentado, las funciones de ponderación introducen un error de amplitud en la magnitud del espectro. Hay un parámetro, conocido como «ganancia coherente» —coherent gain— que indica la ganancia que tiene una cierta ventana con relación a la rectangular [17] y se define como:

$$CG = \frac{1}{N}\sum_{n=0}^{N-1} w[n] \tag{4.54}$$

Donde *CG*, es la ganancia de coherencia, *N*, el número de datos de la ventana, y *w[n]* cada uno de los valores de la ventana.

En cualquier ventana distinta de la rectangular, esta ganancia es menor a la unidad. Para corregir el error en amplitud debida al uso de la función de ventana, se ha de dividir la amplitud de la magnitud del espectro por la ganancia coherente [14]. En la siguiente tabla, se resume las distintas ganancias coherentes para las distintas ventanas utilizadas [13] [17]:

Ventana	CG
Rectangular	1,00
Hanning	0,50
Hamming	0,54
Blackman-Harris	0,36
Bartlett	0,50

Tabla 20: Ganancias Coherentes para las funciones de ventana utilizadas

Para corregir el nivel de una ventana concreta, bastará con dividir cada uno de sus valores por la ganancia de coherencia referente a esa ventana.

En definitiva, de los datos capturados en el dominio del tiempo, solo se utilizan la mitad para la representación de la magnitud del espectro; la frecuencia de cada dato se calcula multiplicando el índice de los datos del registro de la magnitud de frecuencia por la resolución de frecuencia —expresión (4.45) más atrás—; y para la amplitud, se divide cada dato de la magnitud entre la mitad del número de datos del registro en el dominio del tiempo, a excepción del primero —la componente de continua— que se divide por el número de datos capturados. Si se emplea una ventana de ponderación temporal distinta de la Rectangular, se han de dividir los datos de la magnitud espectral por la ganancia de coherencia de la ventana utilizada, además. Se puede utilizar el dato de índice N/2, que representa la frecuencia de Nyquist —teniendo un total de N/2+1 datos para la representación en el dominio de la frecuencia—, en este caso se realizan las mismas correcciones de nivel que para el dato de la componente continua –dato de índice cero—. Para este proyecto no se ha utilizado este.

4.5.4 Descripción de los atributos, tipos (enumeraciones) y métodos de acceso a los atributos de la clase “DispositivoFFT”.

La clase *DispositivoFFT* tiene como misión la captura y procesamiento de los datos. Para llevar a cabo su cometido se vale de una serie de atributos internos —de carácter privado de la clase—, de la definición de varios tipos, de colas de RTOS, de métodos de acceso a los atributos y métodos de carácter funcional. La declaración de tipos es la siguiente:

```
1  class DispositivoFFT
2  {
3      public:
4
5          DispositivoFFT(): //lista de inicialización
6
7          . . .
8
9          ///enumeraciones públicas (ventana en uso y reloj de adquisición)
10         typedef enum
11         {
12             RECTANGULAR=0,
13             HANNING=1,
14             HAMMING=2,
15             BLACKMAN_HARRIS=3,
16             BARLETT=4
17         } TiposVentana;
18
19         typedef enum
20         {
21             FREC36MHZ,
22             FREC18MHZ,
23             FREC12MHZ,
24             FREC9MHZ
25         } RelojMuestreador;
26
27
28     private:
29
30         . . .
31
32         //enumeraciones privadas
33         typedef enum
34         {
35             ESTADO1,
36             ESTADO2
37         } EstadosCaptura;
38
39         . . .
40
41 };
```

Código 4.105: Tipos enumerados de la clase DispositivoFFT

La enumeración existente entre la líneas 10 y 17 —*TiposVentanas*— define los distintos tipos de ventanas disponibles y será usada como tipo para los atributos internos *VentanaActual* y *VentanaAnterior* de esta clase —*DispositivoFFT*— que representan, como sus nombres indican, el tipo de ventana en uso y la ventana que se usó con anterioridad. También será el tipo del atributo *ventana* del modelo —atributo reflejo de *VentanaActual* para poder consultar desde el entorno gráfico la actualmente en uso sin necesidad de hacer la consulta a la instancia *DispositivoFFT* (el atributo ventana del modelo solo se actualiza con el contenido de *VentanaActual* cuando este último cambia en el contexto de la aplicación) —. También sirve de tipo de variables internas a funciones en el presentador y la vista, así como tipo en la información de mensajes de las colas.

El tipo enumerado entre las líneas 19 y 25 —RelojMuestreado— define las diferentes frecuencias de reloj de gobierno del ADC —en configuración triple entrelazada— que no hay que confundir con la frecuencia de muestreo —aunque está íntimamente ligada con ella y será objeto de descripción en la sección 4.5.5—. Este tipo es utilizado en el atributo *muestreador* de esta clase, así como el atributo homónimo del modelo —que es reflejo del anterior para uso interno del contexto gráfico—. También sirve de tipo para la información portada en los mensajes de las colas, como también para realizar conversión de tipos explícita en la vista.

Por último, como definición de tipos, está la enumeración de carácter privado de la clase *DispositivoFFT*, que establece dos estados en los que se puede encontrar la *tareaFFT* —función principal de la clase *DispositivoFFT* encapsulada en una tarea de RTOS—. Esto permite gobernar una máquina de estados —dos estados— donde se alternarán dos arrays para que, mientras uno se use para la captura de datos por parte de la DMA, el otro se mande al entorno gráfico con los datos capturados anteriormente y procesados —cálculo de la FFT, representación logarítmica y escalado gráfico—, todo esto será explicado con más detalle en la sección 4.5.8.

Los atributos que la clase utiliza son:

```
1  class DispositivoFFT
2  {
3      public:
4
5          DispositivoFFT(): //lista de inicialización
6
7          . . .
8
9      private:
10
11         . . .
12
13         //mensajes para colas RTOS
14         Mensaje mensaje_recibido;
15         Mensaje mensaje_emitido;
16
17         //atributos
18         arm_rfft_fast_instance_f32 S;
19         EstadosCaptura estados_captura;
20         bool Ventana;
21         TiposVentana VentanaAnterior;
22         TiposVentana VentanaActual;
23         bool AutoPan;
24         bool AutoAjusteHecho;
25         bool Seguimiento;
26         bool EscalaLogaritmica;
27         uint8_t canal;
28         RelojMuestreador muestreador;
29         bool temporal;
30         float EscalaVertical;
31         float DesplazamientoVertical;
32         bool escalaVerPosVerModificada;
33         uint16_t IndiceInicial;
34         uint16_t IndiceFinal;
35 };
```

Código 4.106: Declaración de atributos de la clase DispositivoFFT

La instancia de la clase *DsipositivoFFT* se comunica con el modelo del contexto gráfico a través de dos colas, una de recepción de mensajes y otra de envío —sección 4.5.9—. Los mensajes de estas colas consisten en unas estructuras que contienen cinco datos miembros: uno de identificación de mensaje y cuatro para posibles datos —sección 4.5.9—. Por ello, esta clase necesita dos variables de este tipo; una de recepción y otra de emisión —declaradas en las líneas 14 y 15—.

En la línea 18 aparece la variable *S* de tipo *arm_rfft_fast_instance_f32*, una estructura necesaria para el cálculo de de la DTF —FFT— donde se encapsula información como la longitud de los datos tratados, los factores de rotación de la FFT para esta longitud o los factores de inversión de bits —comentada en la sección 4.5.2—.

Para el control de la secuencia de captura y procesado de datos se utiliza la variable *estados_captura* de tipo *EstadosCaptura* declarada en la línea 19 —la estructura *EstadosCaptura* ha sido descrita en la porción de código anterior—. De qué manera la utiliza la clase *DispositivoFFT* para realizar esta secuencia será descrita en la sección 4.5.8.

En la línea 20 se declara la variable *Ventana* de tipo booleano cuyo cometido será indicar al método t*areaFFT* si se debe aplicar función de ventana o no. Asimismo, se declaran las variables *VentanAnterior* y *VentanaActual* —líneas 21 y 22—, de tipo *TiposVentana* —tipo comentado en la porción de código anterior—, cuya misión es albergar la ventana en uso actualmente y la usada con anterioridad respectivamente. Cuándo y cómo se toma la decisión de aplicar una ventana y qué tipo de ventana, será descrito en las secciones 4.5.7 y 4.5.11.

Las tres siguientes variables booleanas, *AutoPan*, *AutoAjusteHecho* y *Seguimiento* —líneas 23, 24 y 25— están íntimamente relacionadas. *AutoAjusteHecho* indica al método *tareaFFT* que el autoajuste vertical está hecho —valor verdadero por defecto— y por tanto, no se debe realizar ningún ajuste. Para forzar este ajuste, se debe cambiar el valor de esta variable a falso. Ello sucede cuando el usuario manipula el control gráfico correspondiente y el contexto gráfico informa de ello, cambiando el contenido de esta variable a través de un mensaje mandado por la cola de recepción de la instancia de *DispositivoFFT*. Una vez realizado el ajuste vertical, el método *tareaFFT* pone esta variable con valor verdadero para que en la siguiente iteración no se vuelva a realizar un nuevo autoajuste, a no ser que sea especificado desde el contexto gráfico. *AutoPan*, por su parte, posibilita que la acción de autoajuste se realice exclusivamente con los datos visualizados en pantalla y no con todo el registro. Para ello hace uso de las variables *IndiceInicial* e *IndiceFinal* —líneas 33 y 34 respectivamente—, cuyos valores son asignados con los datos del mensaje de la cola de recepción —mensaje desde el contexto gráfico— cuando dicho mensaje es del tipo de solicitud de datos —ver sección 4.5.9—. Si *AutoPan* tiene un valor verdadero, modifica el comportamiento del autoajuste —realizado con la puesta a falso de la variable *AutoAjusteHecho*— para que este solo tenga en cuenta los valores en el rango visualizado. Por tanto, la variable *AutoPan* no modifica la forma de visualizar los datos por sí misma, sino que predispone a la instancia de *DispositivoFFT* para que el siguiente y sucesivos autoajustes solo lo hagan en la zona visualizada. Es por ello que en el entorno gráfico se deberá disponer de un control referente al autoajuste —que cada vez que se accione produzca el ajuste vertical— y otro para el autopan —que predisponga al autoajuste solo para los datos visualizados—. La variable Seguimiento es similar a *AutoAjusteHecho*, con la diferencia que la primera realiza un autoajuste vertical de forma continúa en cada iteración. Para ello, Seguimiento ha de tener un valor verdadero mientras se desee el autoajuste continuo. Esta propiedad es útil cuando se usa conjuntamente con *AutoPan* en la presentación de los datos de forma lineal —no logarítmica—.

La variable booleana de la línea 26, *EscalaLogaritmica*, sirve precisamente para ese propósito. Si su valor es verdadero, los datos serán presentados en pantalla con escala logarítmica y si es falso, con escala natural. Esta variable debe tener un control dentro del entrono gráfico que actúe directamente sobre ella —claro está, a través de las colas de RTOS—.

La variable entera sin signo de ocho bits, *canal* —línea 27—, alberga el número de canal de entrada a calcular y presentar la FFT. Sus dos posibles valores son 1 y 2, y al igual que las variables anteriores, necesita de un control gráfico que pueda modificar su contenido.

La siguiente variable, *muestreador* —línea 28—, es de tipo *RelojMuestreador*, una enumeración —mostrada en la porción de código anterior— que especifica los posibles valores con los que se puede disciplinar los periféricos ADC del micro STM32F4 —no confundir con frecuencia de muestreo—. Estos son de 36 MHz, 18 MHz, 12 MHz y 9 MHz —que corresponden, aproximadamente[24] a las resoluciones frecuenciales de 3,5156 kHz, 1,7578 kHz, 1,1719 kHz y 0,8789 kHz —ver sección 4.2 y 4.5.5—. El reflejo gráfico de esta variable consistirá en un control en el que se puedan seleccionar uno de estos cuatro valores.

La variable booleana que aparece en la línea 29, llamada *temporal*, posibilita la visualización de los datos temporales capturados —si su valor es verdadero— o procesados mediante la transformada de Fourier —si su valor es falso—. Esto permite mostrar la captura en el dominio temporal —a modo de osciloscopio— o en el dominio de la frecuencia —a modo de analizador de espectros—. En cualquier caso, hay que aclarar que la utilidad de esta variable es puramente de depuración, ya que la pretensión no ha sido realizar un osciloscopio. Los datos temporales capturados no están disparados, puesto que lo que se persigue es la magnitud del espectro, y este es independiente del momento en que se capture la señal —aunque sí tiene algún efecto indeseable como la fuga espectral que se compensa con la utilización de ventanas (sección 4.5.3)—. La consecuencia de ello es la visualización de los datos temporales «en movimiento» cuando la variable *temporal* tiene valor verdadero.

[24] Se ha redondeado la décima de hercio.

Las siguientes tres variables —*EscalaVertical*, *DespelzamientoVertical* y *escalaVerPosVerModificada*, en las líneas 30, 31 y 32 respectivamente — están relacionadas y controlan la visualización vertical de los datos, así como detectar si ha habido modificación en dicha visualización. Las variables *EscalaVertical* y *DespezamientoVerical* tienen sus reflejos en el modelo —variables privadas homónimas en el modelo— que se actualizan cuando se modifican estos valores en la instancia de *DispositivoFFT*. La utilidad de *EscalaVertical* es la de ampliar/atenuar los datos —multiplicarlos por el valor de esta variable— para cambiar su amplitud en la visualización, mientras que la de *DespezamientoVerical* es la de cambiar la posición verticalmente. Estas modificaciones van a tener repercusión en las medidas de amplitud —amplitud por división o medida de cursores— que serán tenidas en cuenta en el contexto gráfico con los valores de las variables homónimas del modelo. Para informar que han sucedido estos cambios, el contexto de la instancia de la clase *DispositivoFFT*, utiliza la tercera variable — *escalaVerPosVerModificada*—, que pone a valor verdadero cuando una de las dos anteriores ha sido modificada, provocando que en al mensaje de la cola de salida —la que realiza la comunicación del contexto de la aplicación hacia el contexto gráfico— contengan la notificación que la escala o la posición vertical han sido modificados en la aplicación —este mensaje contiene también los valores modificados— y deben ser actualizados en el modelo. Esta modificación y la consecuente notificación, pueden ser disparadas tanto por el cambio de la escala o posición vertical en el contexto gráfico —mediante controles en el interfaz gráfico— o mediante la solicitud de autoajuste vertical en el contexto gráfico igualmente.

Las dos últimas variables miembro, *IndiceInicial* e *IndiceFInal* —líneas 33 y 34—, ya han sido comentadas con anterioridad, y se utilizan para realizar el autoajuste sobre los datos visualizados exclusivamente — no sobre todo el registro capturado—. Cada vez que el modelo solicita nuevos datos a la aplicación, a través de un mensaje en la cola —que se realiza en cada tick[25] del entorno gráfico—, se envían los índices inicial y final de los datos que están dentro de la visualización actual. De esta forma la aplicación tiene conocimiento en todo momento qué rango de datos del registro se está visualizando y dónde aplicar el autoajuste si se solicita hacerlo exclusivamente en la zona visualizada —atributo *Autopan*—.

Cada una de estas variables miembro es accedida a través de un par de método; uno para actualizar su valor desde fuera de clase —el nombre del método está precedido por el prefijo «*pon*» seguido del nombre de la variable—, y otro para recuperar su valor —el nombre del método va precedido por el prefijo «*obten*» seguido del nombre de la variable—. Este mecanismo posibilita el control del valor almacenado en las variables internas así como su valor devuelto, constituyendo la característica que se conoce como encapsulamiento en los lenguajes orientados a objetos. En el caso concreto de la clase DispositivoFFT, casi todos los métodos de este tipo se limitan a actualizar y devolver el valor de las variables directamente, pero su utilización posibilita un control de acceso en versiones futuras de la misma. Sin embargo, sí hay dos métodos que hacen uso de esta característica:

[25] Periodo de tiempo de refresco del entorno gráfico que idealmente se corresponderá con el periodo de sincronismo vertical de pantalla, aunque esto varía dependiendo de la carga de la información a visualizar.

```
1  //------------------------------------------------------------------------
2  void DispositivoFFT::ponerEscalaVertical(float escala)
3  {
4      EscalaVertical=escala;
5      escalaVerPosVerModificada=true;
6  }
7  //------------------------------------------------------------------------
8  void DispositivoFFT::ponerPosicionVertical(float posicion)
9  {
10     DesplazamientoVertical=posicion;
11     escalaVerPosVerModificada=true;
12 }
13 //------------------------------------------------------------------------
```

Código 4.107: Métodos de acceso a variables miembro de la clase DispositivoFFT

En este caso, tanto en la actualización de la variable *EscalaVertical* —líneas 2 a 6— como en la actualización de la variable *DesplazamientoVertical* —líneas 8 a 12— se modifica, además, la variable *escalaVerPosVerModificada,* poniéndola a estado verdadero. Desde el punto de vista del programador, es decir, desde fuera de la clase, esto es totalmente transparente, teniendo la impresión que solo se modifican las variables a las que hace referencia el nombre del método. Esta modificación adicional —la variable *escalaVerPosVerModificada*— posibilita que cuando se actualicen los valores tanto de *EscalaVertical* como de *DesplazamientoVertical*, se notifique a la función principal de la clase —*tareaFFT*— que alguno de ellos ha sido modificado, y en consecuencia, se tome la acción oportuna —que el mensaje de salida desde la instancia de la clase *DIspositivoFFT* hacia el modelo del entorno gráfico, contenga la indicación que se han de actualizar la variables homónimas de escala y desplazamiento vertical —.

Existen otros métodos de la clase que solo modifican ciertas variables miembro y no se dispone de método para recuperar su valor. Estas variables pueden ser vistas como de solo escritura y los nombres de los métodos usados para actualizarlas indican la acción que realiza el cambio de la variable. Así, para poner la variable *AutoajusteHecho* a falso —provocar el autoajuste vertical—, se utiliza el método *autoAjustar*; para poner la variable *EscalaLogaritmica* a cierta o a falsa —cambiar la visualización de escala a logarítmica o natural—, se utiliza el método *escalarLogaritmica*; para poner la variable *Seguimiento* a cierta o a falsa —posibilitar el autoajuste vertical continuo o no—, se utiliza el método *seguimientoActivo*; y finalmente para la visualización temporal o frecuencial de los datos, se utiliza el método *verTemporal*.

Hay métodos de carácter privado:

```
1  class DispositivoFFT
2  {
3      public:
4
5          . . .
6
7      private:
8
9          ///métodos privados
10         void crea_hanning(float *array, uint16_t N);
11         void crea_hamming(float *array, uint16_t N);
12         void crea_blackman_harris(float *array, uint16_t N);
13         void crea_bartlett(float *array, uint16_t N);
14
15         void interpretaMensaje(void);
16
17         . . .
18 };
```

Código 4.108: Métodos privados de la clase DispositivoFFT

Los métodos que aparecen en las líneas 10, 11, 12 y 13, crean la función de ventana a la que su nombre hace referencia en el área de memoria especificada en su primer argumento. Estas funciones se multiplican por los

datos capturados para minimizar el efecto de fuga espectral —ver apartados 4.5.3 y 4.5.7—. El método privado interpretaMensaje ya ha sido mencionado y su función es interpretar el contenido de los mensajes entrantes en la aplicación procedentes del entorno gráfico —ver apartado 4.5.11—.

Finalmente están los métodos de inicialización —inicializaFFT y configuraHardware— así como el método principal —tareaFFT— encapsulado en una tarea de FreeRTOS, que serán tratados en las siguientes secciones.

4.5.5 Configuración del ADC —modo triple entrelazado—.

El micro *STM32F429ZIT6* —utilizado en la tarjeta *32F429IDISCOVERY*— posee tres convertidores analógico-digitales —*ADCs*—, cada uno de los cuales con 18 posibles entradas seleccionables. El esquema de cada uno de ellos es el siguiente [4]:

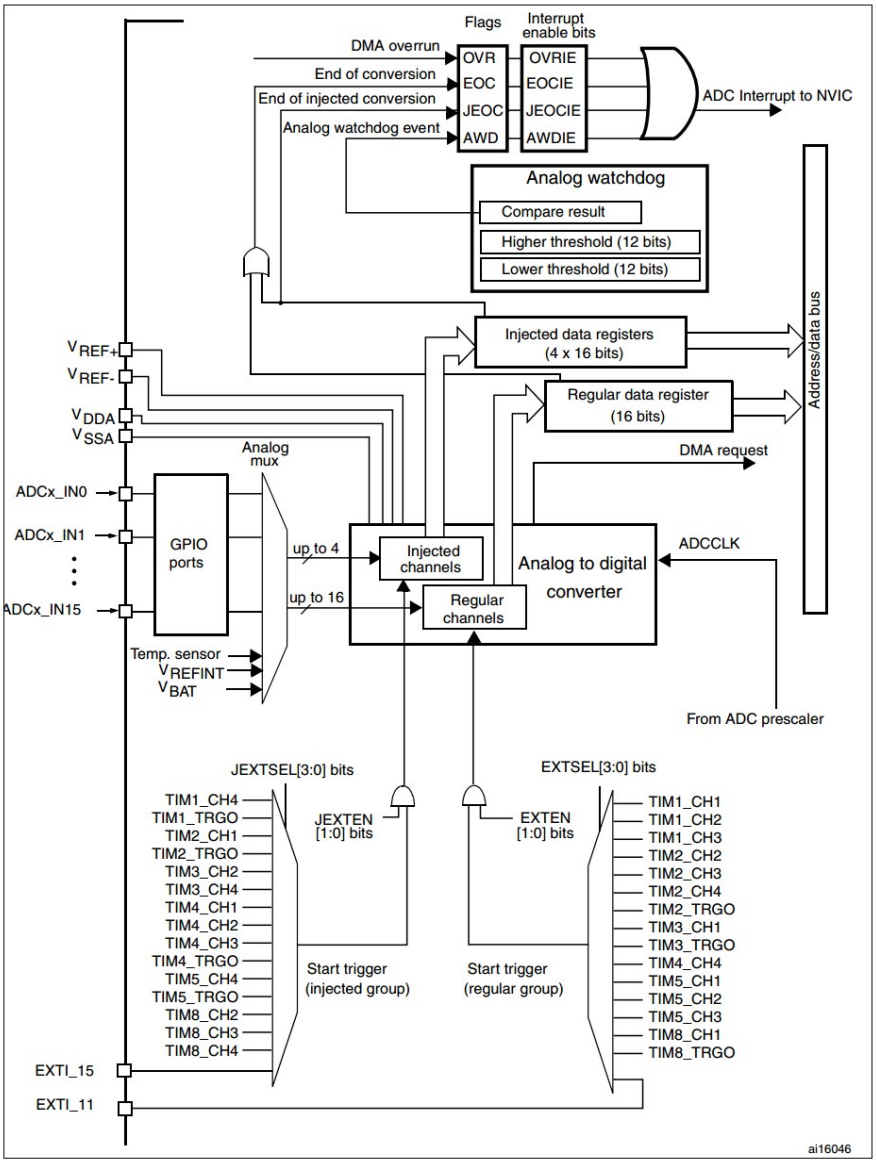


Ilustración 4.50: Esquema ADC del micro STM32F429

De estas entradas, 16 son correspondientes a fuentes externas —ADCx_IN0 a ADCx_IN15— y tres pertenecientes a señales internas —sensor de temperatura, tensión de referencia y tensión de batería—. Aunque, finalmente, el número de canales disponibles en el convertidor va a depender del encapsulado concreto que tenga el micro. Las entradas interesantes para este proyecto son las externas. Como solo se posee un convertidor para varias entradas, es necesario secuenciarlas cuando se pretende realizar la conversión en varias de ellas. Aún así, es posible solo convertir desde una única entrada seleccionada —el caso específico de este proyecto—. Cuando son varias las entradas a convertir, hay la posibilidad de seleccionar si se trata de canales regulares o inyectados. En el caso de canales regulares, se puede configura una secuencia de hasta 16 canales —entre entradas internas y externas— en el orden de conversión deseado. El resultado de la conversión de cada canal es almacenado en un único registro de 16 bits —Regular data register—, por ello se hace necesario la existencia de un mecanismos para almacenar el contenido de este registro en un lugar diferente para no sobrescribir la conversión de un canal con el contenido de la conversión del siguiente —error overrun—. Este mecanismo se trata de una petición DMA después de la conversión de cada canal. Por otro lado, los canales inyectados tienen la particularidad de ser intercalados entre la secuencia de los regulares cuando realizan una conversión —mandada de forma interna o externa mediante señales de disparo—. Por el contrario, solo se puede establecer una secuencia de cuatro canales inyectados cuyas conversiones son almacenadas en cuatro registros —Injected data registers—, uno para cada canal convertido, con la particularidad de no poder utilizar la petición DMA. Tanto para un tipo de canal como para otro, la conversión puede ser iniciada mediante señales hardware —internas, procedentes de otros periféricos, o externas, procedentes de los pines del micro—, o mediante una señal software, poniendo a uno el bit correspondiente del registro de control 2 del ADC —ADC_CR2—, o utilizar las funciones de la librería estándar de periféricos para STM32 —como se mostrará más adelante—. Existen una serie de banderas de estado que indican, entre otros sucesos, el final de conversión de canales regulares —EOC, configurable entre indicar final de conversión de cada canal del grupo a convertir o de todo el grupo—, o error de sobrexcitara en el registro de canales regulares —OVR—. Hay otras banderas que no se comentan, ya que no han sido utilizadas en este proyecto. A cada bandera le corresponde el disparo de una interrupción configurable —para EOC se puede disparar la interrupción EOCIE, o para OVR la OVRIE—. Tanto si se convierte un solo canal como una secuencia de ellos —modo scan—, se puede hacer de modo simple —single—, donde, una vez terminada la conversión, del único canal o la secuencia, la adquisición se detiene y es necesario una nueva señal de disparo —interna, externa o por software—, o de modo continuo —continuous— donde la adquisición no se detiene, volviendo a realizar una nueva conversión tras de la última. Todo el sistema es gobernado por el reloj ADCCLK que proviene de un preescalador que divide la señal de reloj del bus APB2 —PCLK2, ver esquema del reloj del micro en la sección 4.2— entre los valores de 2, 4, 6 y 8 [4]. El máximo reloj admisible —el que posibilita la mayor rapidez de conversión— es de 36 MHz [5] —reloj PCLK2 igual a 72 MHz—, que asegura una velocidad de muestreo de 7,2 MSPS en configuración triple entrelazada, es decir, la frecuencia máxima a muestrear es de 3,6 MHz —teorema de Nyquist—. Para conseguir esta rapidez, no es suficiente con utilizar un único convertidor, sino que se tienen que utilizar los tres que tiene el micro en modo entrelazado —mientras que uno está en fase de conversión, el siguiente ya está en fase de muestreo—. Los tres convertidores comparten las mismas entradas y el inicio de conversión está controlado por el primer ADC. Para ello, se utiliza una configuración individual de cada convertidor y una conjunta del agrupamiento.

En este proyecto se van a utilizar los tres ADCs, que están disponibles el micro, en configuración triple entrelazada, muestreando desde un único canal —con la posibilidad de seleccionarlo entre dos disponibles—, disciplinando el sistema con las frecuencias de 36 MHz, 18 MHz, 12 MHz y 9 MHz —valores del preescalador a 2, 4, 6 y 8—, que corresponden a las frecuencias de muestreo de 7,2 MHz, 3,6 MHz, 2,4 MHz, y 1,8MHz —las resoluciones de frecuencia respectivas son 3,515625 kHz, 1,7578125 kHz, 1,171875 kHz y 0,87890625 kHz (resultado de dividir las frecuencias de muestreo entre 2048, que es el número de datos tratados)—.

Para llevar a cabo esta configuración conjunta de los ADCs, se han de dar los siguientes pasos:

1. Establecer las estructuras de configuración de cada uno de los periféricos internos del micro que están involucrados –que son: el puerto GPIO (donde se localizan las entradas a convertir), el DMA, el NVIC, cada ADC de forma individual y el ADC de forma común—.
2. Habilitar los relojes de los buses de los tres ADC —ADC1, ADC2 y ADC3—, del DMA2 —utilizado en la transferencia de datos adquiridos—, y el puerto GPIOC —que contiene las dos entradas a utilizar en el conjunto convertidor—.
3. Configurar las dos entradas del puerto GPIOC utilizadas en el conjunto convertidor.
4. Configurar el canal y flujo del DMA2 —transferencia de los datos adquiridos a memoria—.
5. Configurar el NVIC para que el DMA2 realice una interrupción al finalizar la transferencia de 2048 datos.
6. Realizar la configuración del conjunto convertidor —configuración común— y la configuración de cada convertidor individual.
7. Configurar cada uno de los dos canales en cada convertidor individual.
8. Habilitar la DMA para el conjunto convertidor, habilitar cada convertidor e iniciar la conversión.

Toda la inicialización del convertidor, se realiza en el método miembro *configuraHardware* de la clase *DispositivoFFT*.

Para declarar las estructuras de configuración de los periféricos, así como habilitar los relojes de los buses, se utiliza la siguiente porción de código del método *configuraHardware*:

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3      /* Se declaran las estructuras de configuración */
4      GPIO_InitTypeDef       GPIO_InitStructure;
5      DMA_InitTypeDef        DMA_InitStructure;
6      NVIC_InitTypeDef       NvicBaseTiempos
7      ADC_InitTypeDef        ADC_InitStructure;
8      ADC_CommonInitTypeDef  ADC_CommonInitStructure
9
10     /* Se habilitan los relojes de los buses de los periféricos */
11     RCC_AHB1PeriphClockCmd( RCC_AHB1Periph_GPIOC , ENABLE);
12     RCC_AHB1PeriphClockCmd( RCC_AHB1Periph_DMA2 , ENABLE);
13     RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC1 , ENABLE);
14     RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC2 , ENABLE);
15     RCC_APB2PeriphClockCmd( RCC_APB2Periph_ADC3 , ENABLE);
16
17    . . .
18
19 };
```

Código 4.109: Estructuras de inicialización del método configuraHardware de la clase DispositivoFFT

Entre las líneas 4 a 8, de la porción del código mostrado, se declaran las estructuras mencionadas en el paso 1 de configuración. Estas estructuras —así como el resto de código del método *configuraHardware*— pertenecen a la librería estándar de periféricos stsw-stm32138 y contienen miembros específicos para poder inicializar el periférico a que hacen referencia. Entre las líneas 11 y 15, se procede a habilitar los relojes de los buses de cada periférico —paso 2 de configuración—. Como se aprecia, tanto el periférico GPIOC como DMA2, pertenecen al bus AHB1, uno de los tres buses que se conectan directamente al CORTEX-M[26], ya que estos dos periféricos necesitan la mayor rapidez posible mientras que los ADCs se conectan al bus APB2 —bus específico para periféricos, con frecuencia de reloj más lenta que el bus AHB—.

Para acometer el paso 3 de configuración, el método *configuraHardware* implementa la siguiente porción de código:

[26] Núcleo del microcontrolador.

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3
4     . . .
5
6    /* Se inicializa la estructura con valores por defecto */
7    GPIO_StructInit(&GPIO_InitStructure);
8
9    /* Se configuran las dos entradas del puerto usadas en las conversión */
10   GPIO_InitStructure.GPIO_Pin = GPIO_Pin_2 | GPIO_Pin_3;
11   GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;
12   GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;
13   GPIO_Init(GPIOC, &GPIO_InitStructure);
14
15    . . .
16
17 };
```

Código 4.110: Configuración de las dos entradas del convertidor

Al principio se carga la estructura de configuración del periférico GPIO —línea 7— con valores por defecto. Este es el primer paso a dar en la inicialización de cualquier periférico. No es estrictamente necesario, pero asegura tener valores iniciales para todos los parámetros de configuración evitando errores. La estructura de configuración se pasa por referencia —se pasa su dirección—, ya que la función GPIOStructInit ha de modificar su contenido. Luego, una vez cargada la estructura con valores por defecto, se establecen aquellos parámetros de la misma que sean necesarios. Así, entre las líneas 10 a 12, se especifica que los pines a configurar van a ser el 2 y el 3, que se tratan de pines analógicos y no van a tener conectadas resistencias de pull-up o pull-down. Finalmente, en la línea 13, el periférico GPIOC —primer argumento de la función GPIO_Init, que inicializa los periféricos de tipo GPIO— se configura con los valores contenidos en la estructura de configuración —GPIOInitStructure, pasada por referencia como segundo argumento—. Para poder determinar qué pines son los utilizados, hay que tener en cuenta cuales están disponibles en el micro, para una configuración triple de los ADC y encapsulado seleccionado, así como que estén disponibles en la placa elegida. Como para el proyecto se ha utilizado la placa 32F429IDISCOVERY, que dispone del micro STM32F429ZIT6 con encapsulado LQFP de 144 pines, es necesario consultar las hojas de datos de ambos. Examinando la hoja de datos del micro, se obtiene la siguiente información [5]:

Pin number								Pin name (function after reset)(1)	Pin type	I / O structure	Notes	Alternate functions	Additional functions
LQFP100	LQFP144	UFBGA169	UFBGA176	LQFP176	WLCSP143	LQFP208	TFBGA216						
17	28	H6	M4	34	J10	37	M4	PC2	I/O	FT	(5)	SPI2_MISO, I2S2ext_SD, OTG_HS_ULPI_DIR, ETH_MII_TXD2, FMC_SDNE0, EVENTOUT	ADC123_IN12
18	29	H7	M5	35	J9	38	L4	PC3	I/O	FT	(5)	SPI2_MOSI/I2S2_SD, OTG_HS_ULPI_NXT, ETH_MII_TX_CLK, FMC_SDCKE0, EVENTOUT	ADC123_IN13

Tabla 21: Pines de configuración ADC triple para encapsulado LQFP144 del micro STM32F429ZIT6

En esta tabla se muestra que para una configuración ADC triple, hay dos posibles entradas que son PC2 y PC3 —pines 2 y 3 del puerto GPIOC—. También informa que estos pines del puerto C, corresponden a los canales 12 y 13 de la topología ADC triple —información que será necesaria en la configuración de los ADCs (punto 7 de la

configuración)—. Igualmente se informa que los pines para el encapsulado LQFP144 son el 28 y 29, pero esto es irrelevante, ya que al micro se accede a través de los pines de los conectores de la placa de desarrollo 32F429IDISCOVERY, y no directamente. Desde el manual de usuario de esta placa, se obtiene la siguiente información [6]:

MCU pin		Board function																	
Main function	LQFP144	System	SDRAM	LCD-TFT	LCD-RGB	LCD-SPI	L3GD20	USB	LED	Puchbutton	ACP/RF	Touch panel	Free I/O	Power supply	CN2	CN3	CN6	P1	P2
PC1	27						CS												13
PC2	28			CSX	CSX	CSX													16
PC3	29																		15

Tabla 22: Pines de configuración ADC triple disponibles en la placa 32F429IDISCOVERY

Los pines 2 y 3 del puerto GPIOC están disponibles en los pines 16 y 15, respectivamente, del conector P2 de la placa de desarrollo. PC3 no es utilizado por ningún periférico externo en la placa, sin embargo PC2 es utilizado por el LCD de 2,4 pulgadas integrado. La señal utilizada en PC2, CSX, pertenece a una de las señales encargadas de gobernar este display mediante interfaz MCU-8080[27] [27]. Pero, ya que el display es sustituido por otro de 5,1 pulgadas que no utiliza esta interfaz —utiliza el periférico interno del micro LTDC—, esta señal —CSX— deja de tener utilidad y el pin PC2 puede ser usado sin problemas.

El siguiente paso de la configuración —paso 4—, es inicializar el periférico DMA utilizado. Para ello es necesario consultar la nota de aplicación AN4031 o el manual de referencia STM32F42xxx, del fabricante ST, donde se especifica información acerca de este periférico. Los datos se extraen de la siguiente tabla [33][4]:

DMA2 request mapping

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	ADC1	SAI1_A(1)	TIM8_CH1 TIM8_CH2 TIM8_CH3	SAI1_A(1)	ADC1	SAI1_B(1)	TIM1_CH1 TIM1_CH2 TIM1_CH3	-
Channel 1	-	DCMI	ADC2	ADC2	SAI1_B(1)	SPI6_TX(1)	SPI6_RX(1)	DCMI
Channel 2	ADC3	ADC3	-	SPI5_RX(1)	SPI5_TX(1)	CRYP_OUT	CRYP_IN	HASH_IN
Channel 3	SPI1_RX	-	SPI1_RX	SPI1_TX	-	SPI1_TX	-	-
Channel 4	SPI4_RX(1)	SPI4_TX(1)	USART1_RX	SDIO	-	USART1_RX	SDIO	USART1_TX
Channel 5	-	USART6_RX	USART6_RX	SPI4_RX(1)	SPI4_TX(1)	-	USART6_TX	USART6_TX
Channel 6	TIM1_TRIG	TIM1_CH1	TIM1_CH2	TIM1_CH1	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	-
Channel 7	-	TIM8_UP	TIM8_CH1	TIM8_CH2	TIM8_CH3	SPI5_RX(1)	SPI5_TX(1)	TIM8_CH4 TIM8_TRIG TIM8_COM

Tabla 23: Canales y flujos para el periférico DMA2

Para el ADC1 —el ADC que disciplina al ADC2 y ADC3 en configuración triple—, el flujo[28] —stream— correspondiente es el cero y el canal que dispara la transferencia es igualmente el cero, ambos, pertenecientes al

[27] Un tipo de Interfaz para mandar información a un display que posea memoria RAM interna encargada de mantener y refrescar la información.

[28] Cada una de las transferencias de datos, entre ocho posibles, desde un periférico a memoria, desde memoria a periférico o desde memoria a memoria.

periférico DMA2. La transferencia se ha de realizar de periférico —desde la configuración ADC triple— a memoria. Se puede apreciar que se podría haber elegido el flujo cuatro con canal cero, igualmente.

La siguiente porción de código realiza la inicialización DMA:

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3
4     . . .
5
6    /* Se inicializa la estructura con valores por defecto */
7    DMA_StructInit(&DMA_InitStructure);
8
9    /* Se configura el flujo 0 del canal 0 del DMA2*/
10   DMA_InitStructure.DMA_Channel = DMA_Channel_0;
11   DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t)&(ADC->CDR);
12   DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t)&registro2;
13   DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory;
14   DMA_InitStructure.DMA_BufferSize = 1024;
15   DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;
16   DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;
17   DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord;
18   DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_HalfWord;
19   DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;
20   DMA_InitStructure.DMA_Priority = DMA_Priority_High;
21   DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;
22   DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;
23   DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single;
24   DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;
25   DMA_Init(DMA2_Stream0, &DMA_InitStructure);
26   DMA_ITConfig(DMA2_Stream0, DMA_IT_TC, ENABLE);
27   DMA_Cmd(DMA2_Stream0, ENABLE);
28
29    . . .
30
31 };
```

Código 4.111: Inicialización DMA del conjunto convertidor

Como anteriormente —paso 3—, lo primero que se realiza es inicializar la estructura de configuración del periférico DMA con valores por defecto —línea 7—. Se especifica que el canal que dispara la transferencia, es el canal cero —línea 10—, acorde con la tabla antes mostrada. La transferencia se va a realizar desde el periférico —configuración ADC triple— a memoria —línea 13—, concretamente, desde la dirección del registro común de datos de la configuración ADC —línea 11— a, inicialmente, la dirección del *registro2* —línea 12—. Este registro, junto con *registro1*, ambos pertenecientes al archivo *DatosEntreTareas.cpp* —sección 4.5.10—, son los dos registros que van a contener toda la captura de datos temporales de forma alternada —mientras uno es utilizado para recoger los datos temporales mediante DMA, el otro, que contiene la captura anterior, es procesado (cálculo de la FFT) y mandado al entorno gráfico para mostrar los datos por pantalla (ver sección 4.5.8)—. Ya que esta transferencia es desde periférico a memoria, la dirección de la fuente de datos —el periférico— ha de ser constante —línea 15—, sin embargo, cada dato capturado ha de ir en direcciones consecutivas en memoria —línea 16—. Cada conversión de un ADC va a consistir en un dato de 8 bits que se guardará empaquetado con la conversión del siguiente ADC —esto será explicado en más detalle a continuación, en el paso de configuración 6—, por ello, el ancho de datos a transferir será de media palabra —16 bits—, tanto desde el periférico como hacia memoria —líneas 17 y 18—. Ya que la captura consiste en 2048 datos de 8 bits –captura temporal—, al empaquetarse de dos en dos, desde el punto de vista de la transferencia, son 1024 datos —línea 14—. La transferencia de los datos se realiza en modo normal —línea 19, en oposición a circular—, lo que significa que una vez realizado el traspaso de 1024 datos de 16 bits, esta se detiene. La prioridad en la transferencia es alta —línea 20—, aunque esta configuración solo tiene sentido cuando se realizan varias transferencias por el mismo periférico DMA —no es el caso—. No se va a utiliza la fifo interna del periférico —líneas de la 21 a la 24—. Finalmente, se especifica que toda esta configuración es para el flujo cero —línea 25—, que se debe realizar una interrupción al finalizar la transferencia de los 1024 datos de 16 bits —línea 26— y que se proceda a la habilitación del flujo cero —línea 27—. El siguiente paso de configuración —paso 5— se acomete con el siguiente código:

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3
4     . . .
5
6    /* solo se trata la prioridad preferente, sin subprioridad */
7    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4);
8    NvicBaseTiempos.NVIC_IRQChannel = DMA2_Stream0_IRQn;
9    NvicBaseTiempos.NVIC_IRQChannelPreemptionPriority = 15;
10   NvicBaseTiempos.NVIC_IRQChannelSubPriority = 0;
11   NvicBaseTiempos.NVIC_IRQChannelCmd = ENABLE;
12
13   /* Se habilita la rutina de atención a la interrupción del flujo 0
14      de la DMA 2 en el NVIC con prioridad 15 */
15   NVIC_Init(&NvicBaseTiempos);
16
17    . . .
18
19 };
```

Código 4.112: Configuración de la interrupción DMA para indicar fin de transferencia de conversión

En la prioridad de la interrupción solo se va a tener en cuenta la prioridad preferente —línea 7—, lo que significa que, de los cuatro bits que se dispone para especificar la prioridad, los cuatro serán utilizados para determinar la prioridad preferente y ninguno para la subprioridad. El valor numérico de la prioridad será de 15 —línea 9—, para ser algo más baja a la del sincronismo vertical utilizado en el entorno gráfico. La interrupción de la transferencia del DMA2 es puesta en marcha en la línea 15.

El inicio del paso 6 de configuración, consiste en el establecimiento de los parámetros comunes del conjunto que forman los tres ADCs trabajando de forma entrelazada. Se debe especificar que el modo de funcionamiento ha de ser entrelazado —no se convierte simultáneamente, sino de forma sucesiva—, cómo se realiza el mismo, de qué manera se lleva a cabo la petición DMA —configurado el flujo y canal DMA en el punto 4, en este, es dónde se determina cuando realizar cada transferencia— y cuál será el reloj que disciplinará el conjunto. Previamente a mostrar el código que posibilita todo esto, es necesario describir cómo se realiza el entrelazado de convertidores y cómo se efectúa la petición DMA. El siguiente esquema cronológico muestra como se realiza el entrelazado de convertidores:

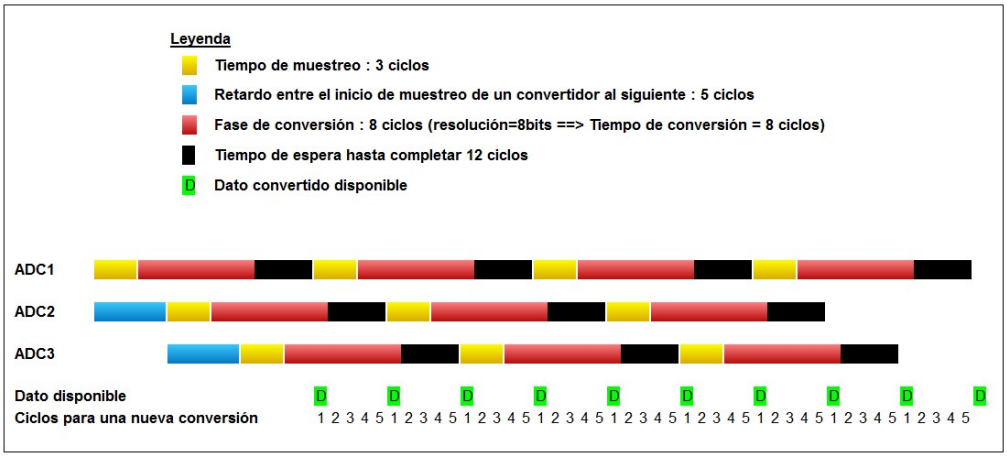


Ilustración 4.51: Esquema cronológico de la configuración ADC triple entrelazada de 8 bits

Para la adquisición de un dato en cada convertidor, se necesita un tiempo de muestreo —donde el dato es retenido en la entrada, mostrado como bloque amarillo en la figura, correspondiente a 3 ciclos de reloj—, configurable entre 3, 15, 28, 56, 84, 112, 144 y 480 ciclos de reloj, y un tiempo de conversión que dependerá del tamaño en bits del dato capturado —ya que se trata de convertidores de aproximaciones sucesivas, este tiempo será igual al número de ciclos de reloj correspondientes al número de bits del dato—, que en la figura se muestra

como un bloque de color rojo que corresponde a 8 ciclos de reloj —8 bits—. Estos dos tiempos son configurados en cada convertidor por separado. En la configuración conjunta se ha de especificar otro tiempo: el tiempo de retardo entre un convertidor y el siguiente para que se produzca el entrelazado. Este tiempo es configurable de 5 a 20 ciclos de reloj —mostrado como recuadro azul en la figura, de anchura 5 ciclos de reloj—. El tiempo de conversión total de cada convertidor, estará compuesto por el tiempo de muestreo más el tiempo de conversión —en el caso de la figura, que corresponde con la configuración real llevada a cabo, es de 11 ciclos de reloj—. Pero para que se produzca el entrelazado, generando datos igualmente espaciados, es necesario un tiempo muerto —no configurable— que asegure un mínimo de 5 ciclos entre muestras entrelazadas. Esto es debido a que la máxima velocidad de muestreo del conjunto triple entrelazado es de 7,2 MSPS[29] [5], y cómo la frecuencia máxima admisible que disciplina el conjunto es de 36 MHz [5], cinco ciclos de este reloj son 7,2 MHz.

La otra cuestión referente a la configuración común, es la petición DMA del conjunto. Para ello se dispone de tres modalidades: DMA modo1, DMA modo2 y DMA modo3. En el modo1, por cada dato convertido se produce una petición DMA. Este modo es idóneo para conversión simultánea de los convertidores —no usado en este proyecto—. Los modos 2 y 3 son similares. Ambos generan una petición DMA por cada dos datos convertidos, lo que implica que ambos datos queden empaquetados juntos en un mismo registro: uno de los datos ocupando la parte baja del registro, y el otro la alta. La diferencia entre ambos modos es que, mientras el modo2 maneja datos 12 bits, el modo3 está pensado para datos de 8 y 6 bits. Por tanto, es este último modo el empleado. La secuencia de cuatro peticiones sucesivas es:

1. ADC_CDR[15:0]=ADC2_DR[7:0]|ADC1_DR[7:0]
2. ADC_CDR[15:0]=ADC1_DR[7:0]|ADC3_DR[7:0]
3. ADC_CDR[15:0]=ADC3_DR[7:0]|ADC2_DR[7:0]
4. ADC_CDR[15:0]=ADC2_DR[7:0]|ADC1_DR[7:0]

El registro ADC_CDR es el registro de datos del conjunto convertidor donde se empaquetan dos muestras sucesivas —correspondientes a dos convertidores sucesivos en la configuración entrelazada—. Se trata de un registro de 32 bits, pero en el modo3 de petición DMA, solo se utiliza su parte baja. El registro ADCx_DR, es el registro de datos individual de cada convertidor —especificado por el número «x»—, y se trata de un registro de 16 bits, aunque en el modo3 solo se utiliza su parte baja. Observando la secuencia de petición, en la primera, se empaqueta el dato del primer convertidor con el segundo; en la segunda petición, el del tercer convertidor y del primero; en la tercera petición, el del segundo convertidor con el tercero; y en la petición cuarta, se vuelve a repetir la secuencia. Este ordenamiento puede parecer un poco artificial, pero hay que tener en cuenta que el direccionamiento del micro es de tipo little endian, y la secuencia anterior —de datos de 16 bits— va a estar alojada en un array que se accederá en formato de 8 bits, haciendo que la secuencia anterior —dentro de este array—, sea: ADC1_DR, ADC2_DR, ADC3_DR, ADC1_DR, ADC2_DR, ADC3_DR, ADC1_DR y ADC2_DR, es decir, se accederá a los datos en el orden en el que han sido adquiridos, tal y como se mostraba en la ilustración anterior. Todo el conjunto estará disciplinado por un reloj proveniente del bus APB2 —reloj PCLK2—, pero a través de un preescalador con los posibles factores de división de 2, 4, 6 y 8, como ya ha sido comentado al principio de esta sección. Ya que el divisor más pequeño es 2 y la frecuencia máxima de entrada al conjunto convertidor es de 36 MHz, la frecuencia ideal del bus APB2 es de 72 MHz —por defecto es de 90 MHz—. Ver sección 4.2.

Visto todo lo referente a la configuración conjunta de los convertidores, se puede mostrar el código que lo implementa:

[29] MSPS.- megamuestras por segundo

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3
4     . . .
5
6     /* ADC configuration común */
7      ADC_CommonStructInit(&ADC_CommonInitStructure);
8      ADC_CommonInitStructure.ADC_Mode = ADC_TripleMode_Interl;
9      ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;
10     ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_3;
11     ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div2;
12     ADC_CommonInit(&ADC_CommonInitStructure);
13
14    . . .
15
16 };
```

Código 4.113: Configuración conjunto convertidor ADC entrelazado

Como en casos anteriores se inicializa la estructura de configuración del conjunto convertidor con valores por defecto —línea 7—. Se establece que la configuración común va a ser triple entrelazada —línea 8—, el modo de petición DMA va a ser el modo3 —línea 10—, el retardo entre convertidores es de 5 ciclos del reloj —línea 9—, y este proviene de dividir el reloj del bus APB2 entre 2 —línea 10—. Finalmente todo ello es activado en la línea 12. En la configuración individual de cada convertidor, se ha de especificar los tiempos de muestreo y conversión —tamaño del dato en bits—, como se explicó anteriormente, cuantos canales de cada convertidor se van a muestrear y cuales, y si la conversión se va iniciar mediante el disparo de una señal —interna o externa— o mediante software. La parte de la función *configuraHardware*, que se encarga de ello, es la siguiente:

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3
4     . . .
5
6      /* ADC1 configuración canales regulares 12 y 13 **********************************/
7      ADC_StructInit(&ADC_InitStructure);
8      ADC_InitStructure.ADC_Resolution = ADC_Resolution_8b;
9      ADC_InitStructure.ADC_ScanConvMode = DISABLE;
10     ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;
11     ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;
12     ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_T1_CC1;
13     ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;
14     ADC_InitStructure.ADC_NbrOfConversion = 1;//solo hay un canal en la secuencia
15     ADC_Init(ADC1, &ADC_InitStructure);
16
17     if (canal==1)
18         ADC_RegularChannelConfig(ADC1, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);
19     else
20         ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
21
22     /* ADC2 configuración canales regulares 12 y 13 **********************************/
23     ADC_Init(ADC2, &ADC_InitStructure);
24     if (canal==1)
25         ADC_RegularChannelConfig(ADC2, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);
26     else
27         ADC_RegularChannelConfig(ADC2, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
28
29     /* ADC3 configuración canales regulares 12 y 13 **********************************/
30     ADC_Init(ADC3, &ADC_InitStructure);
31     if (canal==1)
32         ADC_RegularChannelConfig(ADC3, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);
33     else
34         ADC_RegularChannelConfig(ADC3, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
35
36    . . .
37
38 };
```

Código 4.114: Configuración individual de cada convertidor ADC

Se utiliza la misma estructura de configuración para los parámetros con igual valor en cada convertidor individual. El ancho del dato es de 8 bits, lo que fija el tiempo de conversión —línea 8—; solo se convertirá en un solo canal —líneas 9, modo scan deshabilitado y 14, un solo canal—; la conversión será continua, es decir, cuando se acabe de convertir el canal muestreado, se iniciará una nueva conversión —línea 10—; no se iniciará la conversión por el disparo de una señal ,se hará por software —línea 11—; y el dato adquirido estará alineado en la parte baja del registro de datos de cada convertidor —línea 13—. El parámetro de la línea 12 no es tenido en cuenta ya que no hay señal de disparo, se rellena con este valor para no dejar la estructura sin valor en este campo —aunque con la inicialización con valores por defecto de la línea 7, esto no es estrictamente necesario—. La activación de esta configuración en cada convertidor es llevada a cabo en las líneas 15 —ADC1—, 23 —ADC2— y 30 —ADC3—. Ya que el método *configuraHardware* admite el canal deseado como argumento —con los posibles valores de 1 y 2—, cada convertidor ha de configurarse para reflejarlo. Así, si el argumento es 1, los convertidores han de configurar el canal 12 como entrada —los pines de entrada al micro, conectados a los canales 12 y 13 del conjunto convertidor ya fueron configurados en el puerto GPIOC—, y si es 2, el canal a convertir es el 13. Esto se realiza entre las líneas 17 a 20 para el ADC1, entre las 24 y 27 para el ADC2 y entre las 31 y 34 para el ADC3. En estas mismas líneas, además se especifica que el número de ciclos del tiempo de muestreo de cada convertidor será de 3.

Finalmente, en el método *configuraHardware*, se habilita el conjunto convertidor, cada convertidor de forma individual y se inicia la conversión:

```
1  void DispositivoFFT::configuraHardware(uint8_t canal)
2  {
3
4     . . .
5
6      /* Habilitar petición DMA después de la última transferencia(modo multi-ADC) */
7      ADC_MultiModeDMARequestAfterLastTransferCmd(ENABLE);
8
9      /* Habilitar ADC1 ************************************************************/
10     ADC_Cmd(ADC1, ENABLE);
11
12     /* Habilitar ADC2 ************************************************************/
13     ADC_Cmd(ADC2, ENABLE);
14
15     /* Habilitar ADC3 ************************************************************/
16     ADC_Cmd(ADC3, ENABLE);
17
18     /* Iniciar conversión */
19     ADC_SoftwareStartConv(ADC1);
20    . . .
21
22 };
```

Código 4.115: Habilitaciones de convertidores e inicio de la conversión

Una vez completada toda una transferencia de datos —de 1024 datos de 16 bits o de forma equivalente 2048 datos de 8 bits—, se produce la interrupción de la DMA —del flujo cero—, gestionada por la siguiente rutina de atención a la interrupción:

```
1  extern "C"
2  void DMA2_Stream0_IRQHandler(void)
3  {
4      static signed portBASE_TYPE cambio_contexto = pdFALSE;
5
6      if (DMA_GetITStatus(DMA2_Stream0, DMA_IT_TCIF0))
7      {
8          DMA_Cmd(DMA2_Stream0, DISABLE);//se deshabilita el flujo 0 del DMA 2
9          while (DMA_GetCmdStatus(DMA2_Stream0)!=DISABLE);
10
11         //se borran todas las banderas
12         DMA_ClearITPendingBit(DMA2_Stream0,DMA_IT_TCIF0|DMA_IT_HTIF0|DMA_IT_TEIF0\
13                               |DMA_IT_DMEIF0|DMA_IT_FEIF0);
14         //se notifica que hay datos disponibles desde el DMA
15         xSemaphoreGiveFromISR(semaf_datos_disponibles, &cambio_contexto);
16
17        //se realiza el cambio de contexto desde aquí, si la habilitación del semáforo
18        //anterior habilita una tarea con prioridad mayor a la anterior
19        portEND_SWITCHING_ISR(cambio_contexto);
20     }
21 }
22
```

Código 4.116: Subrutina de atención a la interrupción del flujo cero del DMA2

Esta subrutina, por coherencia, se localiza dentro del archivo fuente de la clase *DispositivoFFT* —DispositivoFFT.cpp—, pero hay que tener presente que no pertenece a esta. De hecho, necesita la especificación de «extern "C"» remarcando que se trata de código C y no de C++, para evitar que el compilador efectúe «mangling»[30] [21] sobre la misma. Una vez disparada la interrupción, en esta rutina se verifica que la causa haya sido por finalización de transferencia de datos —línea 6—. Seguidamente se deshabilita el flujo cero del DMA2 —líneas 8 y 9; esto se realiza para dar una funcionalidad genérica ante futuras transferencias de modo circular—. A continuación se borra la causa de la interrupción —línea 12—. Originalmente solo se ha configura el flujo cero del DMA2 para interrumpir al finalizar la transferencia, pero se procede al borrado de todas las banderas que indican las posibles causas de interrupción para ampliaciones futuras. Si la causa de la interrupción ha sido por finalización de transferencia, se procede a dar el testigo al semáforo *semaf_datos_disponibles*, semáforo declarado en el archivo main.cpp —línea 15—. Esta acción provoca el desbloqueo del método *tareaFFT* de la clase *DispositivoFFT*, que está encapsulado en una tarea del sistema operativo, indicando al mismo que tiene nuevos datos disponibles para ser tratados. Si la habilitación del semáforo provoca que la tarea que se desbloquea es más prioritaria que la tarea que se está ejecutando, se procede al cambio de contexto —línea 19—. Para ello se necesita de la ayuda de la variable *cambio_contexto*, declarada en la línea 4, una variable de tipo booleana del sistema operativo, pasada por referencia al habilitar el semáforo y que es puesta a valor verdadero si es necesario un cambio de contexto.

Alguno de los parámetros configurados en el conjunto triple entrelazado de los ADC, son necesarios ser modificados mediante controles en el entorno gráfico. Tales parámetros son el canal y la frecuencia de reloj del sistema. Por ello, se necesitan sendos métodos en la clase *DispositivoFFT* que posibiliten dichos cambios. Para el caso del cambio de canal, el código del método encargado de llevarlo a cabo —llamado *ponCanal*— es, prácticamente, la reproducción de la porción de código del método configuraHardware encargada de ello. Así el método *ponCanal* se define como:

[30] Técnica que utiliza el lenguaje C++ para renombrar cada función teniendo en cuenta los argumentos y tipos de la misma, posibilitando la sobrecarga.

```
1  void DispositivoFFT::ponCanal(uint8_t canal)
2  {
3      if (canal==1)
4      {
5          ADC_RegularChannelConfig(ADC1, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);
6          ADC_RegularChannelConfig(ADC2, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);
7          ADC_RegularChannelConfig(ADC3, ADC_Channel_12, 1, ADC_SampleTime_3Cycles);
8      }
9      else if (canal==2)
10     {
11         ADC_RegularChannelConfig(ADC1, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
12         ADC_RegularChannelConfig(ADC2, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
13         ADC_RegularChannelConfig(ADC3, ADC_Channel_13, 1, ADC_SampleTime_3Cycles);
14     }
15     this->canal=canal;
16 }
```

Código 4.117: Método ponCanal de la clase DispositivoFFT

Sin embargo, el método encargado de cambiar la frecuencia de reloj que disciplina el conjunto convertidor — *ponFrecMuestreador*—, ha de acceder directamente al registro de configuración común del conjunto, sin utilizar funciones de la librería estándar de periféricos ST ya que no hay ninguna disponible para tal fin. El código de este método es:

```
1  void DispositivoFFT::ponFrecMuestreador(RelojMuestreador muestreador)
2  {
3      ADC->CCR &=~ADC_CCR_ADCPRE;//se borra la información de preescalado en el ADC triple
4      switch (muestreador)
5      {
6          //se graba el nuevo preescalado sin modificar el resto del contenido del CCR
7          case FREC36MHZ:
8              ADC->CCR|=ADC_Prescaler_Div2;
9              break;
10         case FREC18MHZ:
11             ADC->CCR|=ADC_Prescaler_Div4;
12             break;
13         case FREC12MHZ:
14             ADC->CCR|=ADC_Prescaler_Div6;
15             break;
16         case FREC9MHZ:
17             ADC->CCR|=ADC_Prescaler_Div8;
18             break;
19     }
20     this->muestreador=muestreador;
21 }
```

Código 4.118: Método ponFrecMuestreador de la clase DispositivoFFT

Solo se accede a aquellos bits del registro de configuración común —CCR— encargados de contener la información del divisor del preescalador —línea 3—. Dependiendo del valor del argumento pasado —de tipo *RelojMuestreador*, ver sección 4.5.4— así se configura el divisor en estos bits —líneas de la 4 a la 19—. Finalmente, el dato del divisor es almacenado en la variable interna *muestreador* de la clase *DispositivoFFT*.

Ambas funciones no puede ser accedidas directamente desde los controles del entrono gráfico, ya que este está en un contexto de ejecución distinto al del objeto de tipo *DispositivoFFT*. La orden de cambio de alguno los parámetros que estas funciones modifican, llega desde los mensajes enviados por el contexto gráfico a través de las colas. Estos mensajes son interceptados por el método principal de la clase *DsipositivoFFT —tareaFFT*, encapsulada en una tarea del sistema operativo—, que delega la interpretación de los mensajes al método *interpretaMensaje* —otro método de la clase *DispositivoFFT*, ver sección 4.5.4—. Este último método es el encargado de llamar a los distintos métodos del objeto de tipo *DispositivoFFT* acorde con el contenido de los mensajes de las colas, y entre otros, a estos dos, *ponCanal* y *ponFrecMuestreador*.

4.5.6 Implementación de las ventanas temporales.

Las ventanas implementadas en la clase *DispositivoFFT* son: Rectangular, Hanning, Hamming, Blackman-Harris y Bartlett. Tal y como se ha mostrado en el apartado 4.5.4, a cada ventana implementada le corresponde un método —función miembro— de esta clase —a excepción de la Rectangular—. Cada una de estas funciones creará, en un área de memoria, la función de la ventana de ponderación concreta que implementa. Esta región de memoria será compartida por todas las ventanas, ya que solo una de ellas puede ser utilizada a la vez, y se trata del array *ventana*, definido en el archivo DatosEntreTareas.cpp, que será descrito en el apartado 4.5.7.

Teniendo como referencia lo mostrado en la sección 4.5.3, donde se muestran los coeficientes utilizados en las ventanas Hanning, Hamming, Blackman-Harris, se pueden desarrollar las expresiones concretas para implementar cada una de ellas:

$$w_{Han}[n] = 0{,}5 - 0{,}5 \cdot cos\left(\frac{2\pi \cdot n}{N}\right), \quad n = 0{,}1, \dots, N-1 \quad \text{— ventana Hanning—} \tag{4.55}$$

$$w_{Ham}[n] = 0{,}543478 - 0{,}45622 \cdot cos\left(\frac{2\pi \cdot n}{N}\right), \quad n = 0{,}1, \dots, N-1 \quad \text{— ventana Hamming—} \tag{4.56}$$

$$w_{B.Harris}[n] = 0{,}35875 - 0{,}48829 \cdot cos\left(\frac{2\pi \cdot n}{N}\right) + 0{,}14128 \cdot cos\left(\frac{4\pi \cdot n}{N}\right) - 0{,}01168 \cdot cos\left(\frac{6\pi \cdot n}{N}\right), \quad n = 0{,}1, \dots, N-1 \quad \text{— ventana Blackman} - \text{Harris—} \tag{4.57}$$

Así para la ventana Hanning, la función miembro que la implementa es:

```
1  void DispositivoFFT::crea_hanning(float *array, uint16_t N)
2  {
3      float DosPiEntreN = 2*M_PI/(N);
4
5      for(int n=0; n<N; n++)
6          array[n] = (float)(0.5f - 0.5f * arm_cos_f32 (DosPiEntreN * n));
7  }
```

Código 4.119: Implementación de la ventana Hanning en la clase DispositivoFFT

Para permitir mayor versatilidad, esta función, y el resto que implementan las demás ventanas, admite como argumento la dirección del array donde crear la ventana y el número de elementos de la misma. Se declara la variable *DospiEntreN* —línea 3—, ya que es un valor constante utilizado dentro del bucle que crea la ventana —líneas 5 y 6—. *M_PI* es una constante de programa con el valor de la constante matemática π, que está definida al inicio del archivo de cabecera de *DispositivoFFT*.

Para la ventana Hamming, la función que la implementa es:

```
1  void DispositivoFFT::crea_hamming(float *array, uint16_t N)
2  {
3      float DosPiEntreN = 2*M_PI/(N);
4
5      for(int n=0; n<N; n++)
6        array[n] = (float)(0.543478f - 0.456522f * arm_cos_f32 (DosPiEntreN * n));
7  }
```

Código 4.120: Implementación de la ventana Hamming en la clase DispositivoFFT

Y para la ventana Blackman-Harris:

```
1  void DispositivoFFT::crea_blackman_harris(float *array, uint16_t N)
2  {
3      float DosPiEntreN = 2*M_PI/(N);
4      float CuatroPiEntreN = 4*M_PI/(N);
5      float SeisPiEntreN = 6*M_PI/(N);
6
7      for(int n=0; n<N; n++)
8          array[n] = (float)(0.35875f - 0.48829f * arm_cos_f32 (DosPiEntreN * n) + 0.14128f\
9             * arm_cos_f32 (CuatroPiEntreN * n) - 0.01168f * arm_cos_f32 (SeisPiEntreN * n));
10 }
```

Código 4.121: Implementación de la ventana Blackman-Harris en la clase DispositivoFFT

En este caso son necesarias las variables *CuatroPiEntreN* y *SeisPiEntreN* —líneas 4 y 5 respectivamente—, ya que son utilizadas dentro del bucle.

En estas tres porciones de código, para calcular cosenos se utiliza la función *arm_cos_f32*, que es la función coseno de la librería *CMSIS-DSP* y es más rápida en ejecución que otras, como pueda ser la función *cos* de la librería *math.h*. Ello obliga a la inclusión de las cabeceras de la librería *CMSIS-DSP* en el archivo fuente de la clase *DispositivoFFT*, pero esto era necesario con anterioridad, ya que esta clase necesita realizar el cálculo de la FFT que está implementada en esta librería. Para el caso de la ventana Bartlett, la expresión a utilizar se mostró en el apartado 4.5.3, y se reproduce aquí por comodidad:

$$w[n] = \begin{cases} \dfrac{n}{N/2} = \dfrac{2n}{N}, & n = 0,1,\dots,\dfrac{N}{2} \\ w[N-n] = \dfrac{N-n}{N/2} = 2 - \dfrac{2n}{N}, & n = \dfrac{N}{2},\dots,N-1 \end{cases} \tag{4.58}$$

Lo que se traduce en el siguiente código:

```
1  void DispositivoFFT::crea_bartlett(float *array, uint16_t N)
2  {
3      uint16_t mitad = N/2;
4
5      for(int n=0; n<N; n++)
6      {
7          if(n<=mitad)
8              array[n] = 2.0f*n/(N);
9          else
10             array[n] = 2 - 2.0f*n/(N);
11     }
12 }
```

Código 4.122: Implementación de la ventana Bartlett en la clase DispositivoFFT

Estas cuatro funciones no son llamadas directamente sino que se hace a través de la función *ponVentana* de la clase *DispositivoFFT*:

```
1  void DispositivoFFT::ponVentana(TiposVentana vent)
2  {
3      if((vent==VentanaAnterior) && vent !=RECTANGULAR)
4          Ventana=true;
5      else if(vent==RECTANGULAR)
6          Ventana=false;
7      else
8      {
9          switch(vent)
10         {
11             case RECTANGULAR:
12                 break;
13             case HANNING:
14                 crea_hanning(ventana,2048);
15                 break;
16             case HAMMING:
17                 crea_hamming(ventana,2048);
18                 break;
19             case BLACKMAN_HARRIS:
20                 crea_blackman_harris(ventana,2048);
21                 break;
22             case BARLETT:
23                 crea_bartlett(ventana,2048);
24                 break;
25         }
26         Ventana=true;
27     }
28     VentanaAnterior=VentanaActual;
29     VentanaActual=vent;
30 }
```

Código 4.123: Código para la gestión de ventanas de la clase DispositivoFFT

La función mostrada en esta porción de código —función *ponVentana*—, tiene como misión rellenar un array —*ventana*— con 2048 datos correspondientes a la ventana deseada. Este array será el que contenga la función de ventana en uso y será el único que interactúe con el proceso de cálculo de la FFT. Para ello, la función mostrada admite como argumento un dato de tipo *TiposVentana*; una enumeración declarada en la clase *DispositivoFFT* que puede tomar cada uno de los valores de la ventana a que hace referencia —tal y como se mostró en el apartado 4.5.4—. La clase *DispositivoFFT* dispone de dos atributos privados, *VentanaActual* y *VentanaAnterior*, ambos de tipo *TiposVentana*, cuya misión es la de optimizar la creación de las ventanas dentro del array *ventana*. Así, el atributo *VentanaActual* tendrá siempre el valor del argumento pasado a la función, mientras *VentanaAnterior* tendrá el valor anterior de *VentanaActual* —líneas 28 y 29—. De esta forma se puede evitar volver a crear la función de ventana, dentro del array, si la ventana deseada ya es la existente dentro del mismo. Este control se ejerce en la primera sentencia condicional —línea 3— de tal forma que si la ventana solicitada es la que ya está en uso, no se vuelve a crear. La clase *DispositivoFFT* dispone de un atributo de tipo booleano llamado *Ventana*, cuya misión es la de controlar si se aplica la función de ventana a los datos temporales capturados en el ADC. Este discernimiento es necesario ya que, cuando se seleccione como ventana en uso la ventana *Rectangular*, realmente no se aplicará función de ventana. Por ello, en la sentencia de control de la línea 3, aparte de verificar si la ventana solicitada es la que ya está en uso, verifica que no se trate de la *Rectangular*. Si la ventana solicitada es la *Rectangular* —línea 5— se pone a *false* el valor del atributo *Ventana*, indicando a la instancia de la clase *DispositivoFFT* que no debe multiplicar los datos temporales por ninguna función. En caso de ser solicitada el uso de una ventana que no es la actual y no se trata de la *Rectangular*, entra en funcionamiento la sentencia *else* de la línea 7 así como el bloque *switch-case* entre las líneas 9 y 25. Acorde con el argumento pasado a la función, este bloque creará la ventana solicitada llamando a las funciones de creación de ventana antes mostradas. En cualquiera de los casos, pondrá con valor verdadero el atributo *Ventana*, indicando que se aplique la ventana creada a los datos temporales.

4.5.7 Datos compartidos entre contexto gráfico y contexto de la aplicación.

Para la comunicación entre los dos contextos —el del entorno gráfico y el de la aplicación—, se utilizan diversos elementos como son: las colas del sistema operativo, la memoria compartida y los mensajes estructurados en forma de eventos —que viajan a través de las colas—. Cómo todos estos han de ser accedidos por ambos contextos, es obligatorio que estén declarados —y también definidos— en un lugar ajeno a ellos. Por un lado, está el archivo fuente *main.cpp*, donde se declaran y definen las colas, y el archivo de cabecera *ComunicaTareas.hpp*, donde se declaran externamente para ser accedidas desde fuera el archivo main.cpp, así como la estructura de los mensajes. Por otro lado, está el archivo fuente *DatosEntreTareas.cpp*, donde se declaran los arrays de memoria a ser utilizados como memoria compartida entre contextos —además de contener otras declaraciones de arrays utilizadas en la clase *DispositivoFFT* (ver siguiente sección)— y el archivo de cabecera *DatosEntreTareas.hpp*, donde se declaran estos arrays de forma externa para ser accedidos desde fuera del archivo *DatosEntreTareas.cpp*, así como la definición de constantes que determinan sus tamaños. Es por ello, que tanto el archivo de cabecera *ComunicaTareas.hpp*, como *DatosEntreTareas.hpp*, han de estar incluidos en aquellas partes de los contextos que manejen estos elementos.

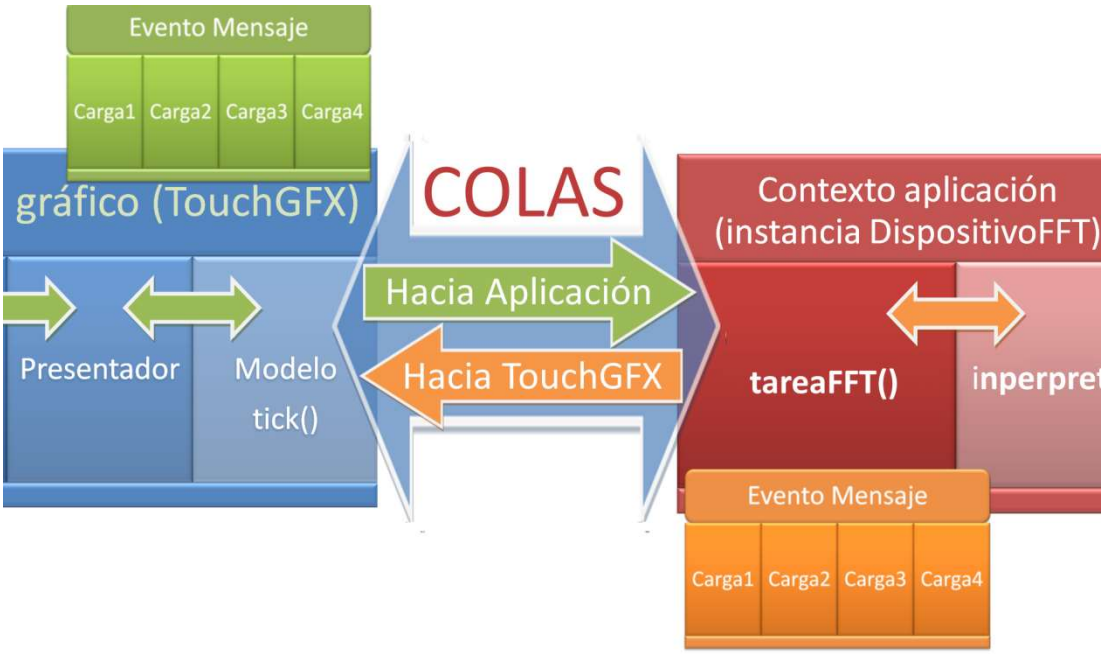


Ilustración 4.52: Colas del sistema entre contextos

Observando la figura anterior, estas partes son el modelo, desde el contexto del entrono gráfico *TouchGFX*, y el método *tareaFFT* de la clase *DispositivoFFT*, desde el contexto de la aplicación. Por tanto, los archivos de cabecera antes nombrados, estarán incluidos tanto en el archivo de cabecera del modelo —*Model.hpp*—, como en el archivo de cabecera de la clase de la aplicación— *DispositivoFFT.hpp*—.

Ya que la comunicación ha de ser bidireccional, del entrono gráfico hacia el contexto de la aplicación y viceversa, se necesitan sendas colas, cada una en un sentido, tal y como se muestra en la ilustración. Ambas manejan el mismo tipo de mensaje consistente en una estructura con cuatro posibles datos y un especificador de

mensaje o tipo evento que se envía de un contexto a otro. La definición y declaración de las colas se realiza de la siguiente forma en el archivo *main.cpp*:

```
1  . . .
2  #include <gui/common/ComunicaTareas.hpp>
3  . . .
4  //Cola para enviar datos desde la tarea de la fft al interfaz gráfico
5  xQueueHandle colaDeAplicacion_a_GUI;
6  //Cola para enviar datos desde el interfaz gráfico a la tarea de la fft
7  xQueueHandle colaDeGUI_a_Aplicacio;
8  . . .
9  int main (void)
10 {
11     . . .
12     colaDeAplicacion_a_GUI = xQueueCreate( 4, sizeof(Mensaje));
13     colaDeGUI_a_Aplicacion = xQueueCreate( 4, sizeof(Mensaje));
14     . . .
15
16 }
```

Código 4.124: Declaración y definición de colas en el archivo fuente main.cpp

Primeramente se declaran las colas para cada uno de los sentidos, *colaDeAplicación_a_GUI*, para el sentido de aplicación hacia entorno gráfico —línea 5— y *colaDeGui_a_Aplicacion*, para el sentido de entorno gráfico hacia aplicación. Luego se han de definir, es decir, qué tamaño tendrá cada una y qué tipo de elemento manejará. Ambas van a manejar como datos a enviar por la cola, estructuras de tipo *Mensaje* —que será explicada más adelante en esta sección—, y sus tamaños serán de cuatro elementos —líneas 12 y 13—. El tamaño es especialmente crítico para la cola que va desde el entorno gráfico hacia la aplicación, pero con cuatro elementos se ha comprobado, experimentalmente, que no produce problemas —incluso se podría utilizar con menos elementos—. Para que estas colas puedan ser usadas externamente al archivo *main.cpp*, se declaran bajo este especificador —*extern*— en el archivo *ComunicaTareas.hpp*, de esta forma, incluyendo este archivo, tanto en el modelo como en la instancia de la clase *DispositivoFFT*, pueden ser accedidas por ambos:

```
1  #ifndef COMUNICACION_TAREAS_HPP
2  #define COMUNICACION_TAREAS_HPP
3  . . .
4  extern xQueueHandle colaDeAplicacion_a_GUI;
5  extern xQueueHandle colaDeGUI_a_Aplicacion;
6  . . .
7  #endif //COMUNICACION_TAREAS_HPP
```

Código 4.125: Declaraciones externas de las colas en el archivo de cabecera ComunicaTareas.hpp

Este sistema permite enviar información consistente en cuatro datos —los contenidos en un mensaje—, pero si se necesita enviar más información, se debe disponer de otro mecanismo. De hecho, la principal actividad de la instancia de la clase *DispositivoFFT* es el cálculo de la magnitud del espectro realizada sobre una captura temporal, y este es de 1024 datos. Para ello se va a disponer de dos regiones de memoria que serán accedidas por ambos contextos. Estas regiones son los arrays *registro1* y *registro2*. El método *tareaFFT* —de la clase D*ispositivoFFT*— va a utilizar ambos registros de forma alternada para realizar el cálculo de la transformada discreta de Fourier, posibilitando que el registro sobre el que se acaba de realizar este cálculo, sea mandado al contexto gráfico para su representación en pantalla, mientras que el otro, sea utilizado para el nuevo cálculo. La forma que tiene la aplicación de informar al entorno gráfico que hay disponibles nuevos datos calculados, es a través de un mensaje —que se especifique en el tipo de evento del mensaje—, pero en este solo se portan cuatro

datos. La solución es utilizar uno de estos datos como puntero que apunte al array que se desea que el entrono gráfico utilice, tal y como se muestra en el esquema de la siguiente ilustración:

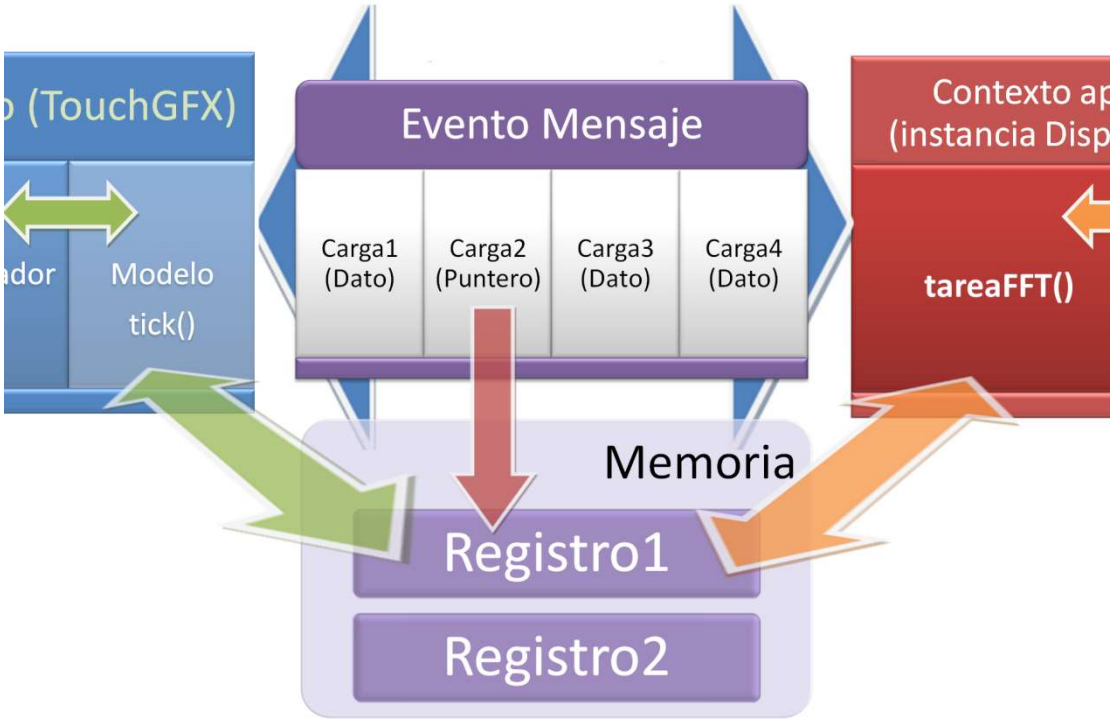


Ilustración 4.53: Memoria compartida entre contextos

Así, cuando *tareaFFT* realice el cálculo de la FFT sobre registro1, bastará que especifique en el campo *Evento* del mensaje, que hay disponible nuevos datos —evento *DATOS_ESPECTRO*— y utilice uno de los cuatro datos libres del mensaje —en el caso de la figura se utiliza el dato *carga2*— para contener la dirección de *registro1*. Cuando el modelo descargue el mensaje de la cola, inspeccionará primero de qué tipo de mensaje se trata —*DATOS_ESPECTRO*—, lo que le informará qué debe hacer y qué contiene cada uno de los datos que vienen con el mensaje. Luego recogerá la *carga2* y la entenderá como dirección de memoria a los datos a representar en pantalla. Mientras tanto, *tareaFFT* usará *registro2* para realizar el nuevo cálculo de la FFT, y una vez concluido, pasará su dirección en un nuevo mensaje hacia el contexto gráfico. Este proceso se repetirá de forma indefinida.

Ambos registros están declarados en el archivo fuente *DatosEntreTareas.cpp*:

```
1  #include <gui/common/DatosEntreTareas.hpp>
2
3  float flotante[N_DATOS_USADOS];
4  float flotante2[N_DATOS];
5  float ventana[N_DATOS_USADOS];
6
7  uint8_t registro1[N_DATOS];
8  uint8_t registro2[N_DATOS];
```

Código 4.126: Arrays usados como memoria compartida, declarados en DatosEntreTareas.cpp

El tamaño de ambos arrays está definido por la constante *N_DATOS*, que a su vez lo está en el archivo de cabecera *DatosEntreTareas.hpp*. En este último archivo también está definida la constante *N_DATOS_USADOS* que fija los tamaños de otros arrays usados exclusivamente en la clase *DispositivoFFT*. El contenido de *DatosEntreTareas.hpp* es:

```
1  #ifndef DATOS_ENTRE_TAREAS_HPP
2  #define DATOS_ENTRE_TAREAS_HPP
3
4  #include <touchgfx/hal/HAL.hpp>//para utilizar los tipos int8 t, uint8 t, int16_t, etc.
5  #include "stm32f4xx.h"//para poder utilizar la librería estándar de STM32
6
7  #define N_DATOS 4096
8  #define N_DATOS_USADOS N_DATOS/2  //2048
9
10 extern uint8_t registro1[N_DATOS]; //2048 datos enteros (para dos canales de 2048 datos)
11 extern uint8_t registro2[N_DATOS]; //2048 datos enteros (para dos canales de 2048 datos)
12
13 extern float flotante[N_DATOS_USADOS]; //2048 datos reales(entrada real para la FFT)
14 extern float flotante2[N_DATOS]; //2048 datos complejos (equivale a 4096 reales)
15                                  //(salida de 2048 datos complejos de la FFT)
16 extern float ventana[N_DATOS_USADOS];
17
18 #endif //DATOS_ENTRE_TAREAS
```

Código 4.127: Arrays usados como memoria compartida

En este archivo se define *N_DATOS* —a 4096— como los datos a ser tratados en el espectro, que equivalen a 2048 números complejos—ya que cada complejo está constituido por su parte real y su parte imaginaría—. Para los arrays *registro1* y *registro2*, este tamaño es el usado, ya que inicialmente se pensó en ellos como áreas de memoria que albergasen dos canales —datos alternados de dos canales debido a que así es como son capturados al utilizar DMA con dos ADCs trabajado de forma simultánea—. Los arrays *flotante*, *flotante2* y *ventana*, son exclusivamente utilizados en el método tareaFFT, pero son declarados externamente aquí —y declarados originalmente en el archivo fuente *DatosEntreTareas.cpp*— ya que, de esa forma se centraliza todas las regiones de memoria usadas por la aplicación. Para los arrays *flotante* y *ventana*, los tamaños son fijados a 2048 datos de tipo flotante. En el primer caso, porque se trata de la captura temporal convertida de dato entero a real o se trata de la magnitud del espectro. En el segundo, porque contiene la ventana de ponderación a ser aplicada a los datos temporales —y ha de ser de igual tamaño al del array de datos temporales—. El array flotante2 tiene un tamaño de 4096 datos, ya que contendrá el espectro complejo. Este archivo de cabecera ha de estar incluido tanto en el archivo *DispositivoFFT.hpp* como en *Model.hpp* —ambos contextos—.

Como se intuye, la carga de datos del mensaje debe ser muy versátil y poder contener varios tipos de datos. Un mensaje, enviado por una cola, debe constar de un identificador del mismo —un elemento que indique al receptor del mensaje qué es lo que está recibiendo—, que se podría denominar como tipo de evento. También debe contener una carga útil de datos con los que el receptor pueda trabajar. El número de estos datos ha de ser lo suficientemente grande como para enviar la información necesaria para cada tipo de mensaje —tipo de evento—, pero lo suficientemente pequeño para no mover grandes cantidades de memoria por las colas. Para este desarrollo se decide crear la estructura del mensaje con cuatro datos. Con lo que el mensaje tendrá que contener, finalmente, cinco elementos: un identificador y cuatro datos. El identificador no podrá ser cualquier valor, será una serie de ellos fijados previamente para poder establecer el acuerdo necesario entre emisor y receptor del mensaje que posibilite las acciones a tomar —por parte del receptor— y qué significado tienen cada uno de los datos contenidos en el mismo así como el tipo de cada uno de ellos. El identificador será, por tanto, un tipo enumerado donde cada elemento del mismo signifique una acción a tomar por parte del receptor. Para el tipo de la carga de datos se utilizará una unión, ya que esto posibilitará que cada uno de ellos pueda ser de diferentes tipos —sin tener que prefijarlos—. La siguiente ilustración muestra lo comentado:

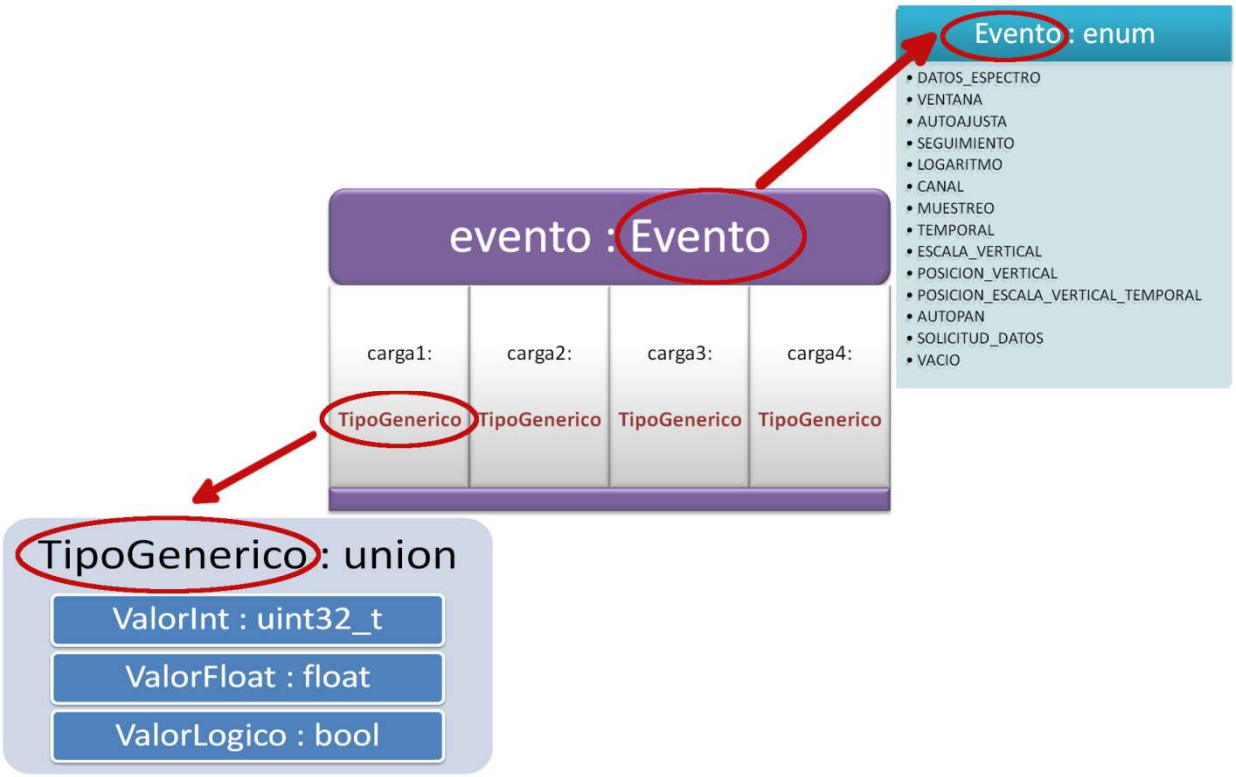


Ilustración 4.54: Estructura de los mensajes —eventos— entre contextos

El mensaje es una estructura —parte central de la ilustración— que contiene el evento —identificador del mensaje— que es de del tipo enumerado *Evento* —parte derecha de la ilustración— y los cuatro datos que son de tipo *TipoGenerico* —parte izquierda de la ilustración—. La enumeración del tipo de evento contiene todos los posibles mensajes que pueden mandarse por las colar —tanto en un sentido como en otro—, mientras que la unión del tipo de los datos, contiene los tres posibles tipos que pueden ser albergados en el mensaje. El código que implementa la estructura del mensaje es la siguiente —contenida en el archivo ComunicaTareas.hpp—:

```
1  #ifndef COMUNICACION_TAREAS_HPP
2  #define COMUNICACION_TAREAS_HPP
3  . . .
4  typedef struct TMensaje
5  {
6      uint32_t evento;
7      TipoGenerico carga1;
8      TipoGenerico carga2;
9      TipoGenerico carga3;
10     TipoGenerico carga4;
11 }Mensaje;
12
13 #endif //COMUNICACION_TAREAS_HPP
```

Código 4.128: Definición de la estructura de los mensajes en el archivo ComunicaTares.hpp

El nombre de la estructura es *Mensaje*, así que cuando se declaren las variables que representen a los mensajes, en la clase DispositivoFFT —ya visto en la sección 4.5.4—, han de ser de este tipo. El primer elemento de la estructura es el tipo de mensaje —*evento*, en la línea 6—. Contrariamente a lo que se podría pensar, de acuerdo con la ilustración anterior, no es del tipo *Evento* sino que es de tipo entero de 32 bits sin signo. Esto es debido a que una enumeración es realmente un tipo entero, pero más restrictivo. El compilador no permite tener valores diferentes a los establecidos en una enumeración, aunque sean válidos desde el punto de vista de un tipo entero. Por ello, se decide establecer este primer elemento de la estructura como *uint32_t*, ya que ello permite

contener los valores de una enumeración, así como cualquier otro valor entero, lo que posibilita la especificación numérica de nuevos tipos de mensajes para ampliaciones futuras —incluso el envío de un dato entero, que represente una información—. Por otro lado, como se mostrará más adelante, el campo *evento* de la estructura, es realmente una agregación de diversa información, y si se define como tipo enumerado, impide esta funcionalidad. Los siguientes cuatro elementos de la estructura son la carga útil del mensaje. Son uniones *TipoGenerico* —que sí concuerda con lo mostrado en la ilustración—.

La enumeración que especifica los posibles mensajes, así como otra información que puede ir empaquetada en esta especificación, se muestra en la siguiente porción de código del archivo ComunicaTareas.hpp:

```
1  //MASCARAS PARA LOS EVENTOS EN LA COMUNICACION ENTRE TAREAS
2  #define MASCARA_TIPO_EVENTO    0xFF000000 //Posibilita extraer el tipo de evento
3  #define MASCARA_TIPO_SUBEVENTO 0x00FF0000 //Posibilita extraer el tipo de subevento
4  #define MASCARA_DATO_EVENTO    0x0000FFFF //Posibilita extraer el dato del evento
5
6
7  //TIPOS DE EVENTOS EN LA COMUNICACION ENTRE TAREAS
8  typedef enum TEvento
9  {
10     //Eventos desde la tarea hardware fft hacia el modelo
11     DATOS_ESPECTRO=(1<<24UL),
12     //Eventos desde el modelo hacia la tarea hardware fft
13     VENTANA=(2<<24UL),
14     AUTOAJUSTA=(3<<24UL),
15     SEGUIMIENTO=(4<<24UL),
16     LOGARITMO=(5<<24UL),
17     CANAL=(6<<24UL),
18     MUESTREO=(7<<24UL),
19     TEMPORAL=(8<<24UL),
20     ESCALA_VERTICAL=(9<<24UL),
21     POSICION_VERTICAL=(10<<24UL),
22     POSICION_ESCALA_VERTICAL_TEMPORAL=(11<<24UL),
23     AUTOPAN=(12<<24UL),
24     SOLICITUD_DATOS= (80<<24UL), //evento por defecto desde la tarea grafica hacia la
25                                  //aplicación
26     //Resto
27     VACIO=0x00000000
28 }Evento;
29
30 typedef enum TSubEvento
31 {
32     //Calificadores de eventos (subeventos) desde la tarea hardware fft hacia el modelo
33
34     //subevento calificador del evento DATOS_ESPECTRO
35     ESCALA_VER_DES_VER_MODIFICADOS=(1<<16UL)
36
37     //Calificadores de eventos (subeventos) desde el modelo hacia la tarea hardware fft
38     //NINGUNO EN LA ACTUALIDAD
39 }SubEvento;
40
41 typedef enum TDatoEvento
42 {
43   //Datos predeterminados que pueden ir empaquetados en el campo "evento" del mensaje
44   //(utilizado como notificación de retorno desde la tarea de la aplicación hacia la tarea
45   //grafica)
46     DATO_RECIBIDO=1
47 }DatoEvento;
```

Código 4.129: Enumeración de tipos de mensajes y enumeraciones accesorias en el archivo ComunicaTares.hpp

La enumeración que especifica los posibles tipos de eventos —mensajes— que pueden ser enviados por las colas, se muestra entre las líneas 8 a 28 del código anterior. Solo existe un tipo de mensaje que se envía desde el contexto de la aplicación hacia el contexto del entrono gráfico: *DATOS_ESPECTRO*. Este tipo de mensaje indica al modelo —contexto del entorno gráfico— que hay datos disponibles para ser representados, y además, el modelo debe entender que en la *carga1* está el valor de la dirección de memoria de los datos a representar —cuya dimensión máxima es de 2048 y que se trata de tipos enteros de 8 bits—; que en la *carga2* está el valor del desplazamiento vertical —posición vertical de la representación, dato de tipo flotante—; que en la *carga3* está la escala vertical de la gráfica —factor por el que han sido multiplicado los datos, de tipo flotante—; y que en la carga4 está el canal utilizado —ver apartado 4.5.5—, aunque este dato en la actualidad no es utilizado por el

entorno gráfico —el modelo maneja su propia información acerca del canal—. Ya que solo hay un tipo de mensaje desde la aplicación hacia el entorno gráfico, se podría ahorrar la especificación del mismo, utilizando el campo reservado al tipo de mensaje, en la estructura *Mensaje*, como un dato más, pero esto impediría ampliar los tipos de mensajes en versiones futuras en este sentido de la comunicación. El resto de mensajes, son utilizados en el sentido de la comunicación desde el entorno gráfico a la aplicación, y serán tratados en el apartado 4.5.9, pero básicamente indican a la aplicación, las órdenes enviadas por el modelo que se han de cumplir. Así por ejemplo, el mensaje de tipo *VENTANA*, indica a la aplicación que el modelo está solicitando la aplicación de un tipo de ventana concreto, especificado en su *carga1*, y debe entenderse este dato de tipo *TiposVentana* —ver apartado 4.5.4—. Existe un tipo de mensaje llamado *SOLICITUD_DATOS*, que el modelo envía a la aplicación cuando únicamente desea que esta última mande los datos que acaba de procesar. En este sentido, este tipo de mensaje es el usado por defecto, y el modelo lo lanza a la aplicación cuando no tiene una orden particular que mandarle. Cualquier otro mensaje desde el modelo, además de instruir a la aplicación para realizar una acción concreta, implícitamente está solicitando los últimos datos.

El elemento *evento* de la estructura *Mensaje*, no solo va a contener la información de la enumeración *Evento*, sino que va a contener otras dos informaciones más: un posible calificador del evento o subevento y un dato muy relacionado con el evento. Para ello se subdivide los 32 bits que conforman el campo *evento* de la estructura *Mensaje*, de tal forma que los bits de las posiciones 24 a 31 se reservan al tipo de evento —contendrá uno de los valores definidos en la estructura *Evento*—, los bits de las posiciones 16 a 23 se reservan para el subevento y los bits del 0 al 15 para el dato. Para que ello sea posible es necesario que la información que se vaya a albergar en estas subdivisiones tenga en cuenta esta distribución de bits. Así, los valores de los elementos de la enumeración *Evento* están desplazados hasta el bit 24. Para los subeventos se crea la enumeración *SubEvento* —líneas de la 30 a la 39 del código anterior—, con un solo elemento, *ESCALA_VER_DES_VER_MODIFICADOS*, cuyo valor ha sido desplazado hasta el bit 16 para encajar en la distribución de bits hecha sobre el elemento *evento* de *Mensaje*. Este subevento califica al evento *DATOS_ESPECTRO*, que informa al modelo, que además de contener el puntero de los datos procesados en *carga1*, los datos de posición y escala verticales en *carga2* y *carga3*, respectivamente, son nuevos debido a que la aplicación los ha modificado —tras un autoajuste, por ejemplo—, y deben ser actualizados en los atributos reflejo en el modelo, ya que los utiliza para determinar los valores de amplitud de los datos presentados. Por otro lado se define la enumeración *DatoEvento* —en las líneas 41 a 47—, con su único elemento *DATO_RECIBIDO*, que sirve para indicar al modelo, desde la aplicación, que el último mensaje desde el modelo ha sido recibido y procesado. Este dato de evento se envía conjuntamente con el evento *DATOS_ESPECTRO* hacia el modelo. El modelo, además de identificar el evento —bits 24 al 31— y el calificador de evento —bits 16 al 23—, extrae el posible dato que viene en el campo *evento* del *Mensje* —bits 0 al 15—. Si este dato de evento es *DATO_RECIBIDO,* el siguiente mensaje que mandará el modelo a la aplicación será el por defecto: *SOLICITUD_DATOS*, a no ser que se sobrescriba debido a una acción en la vista —como, por ejemplo, presionar en el botón para utilizar la ventana Hanning—. Para poder acceder a los distintos bits que conforman la división del elemento *evento* de la estructura *Mensaje*, se definen tres constantes en las líneas 2, 3 y 4 del código anterior. Estas tres constantes sirven como máscaras para extraer cada conjunto de bits mediante la operación *and* lógica. Finalmente queda especificar el tipo de los dato de la carga útil, que tal y como se mostraba en la ilustración, se trata de una unión cuya definición es:

```
1  #ifndef COMUNICACION_TAREAS_HPP
2  #define COMUNICACION_TAREAS_HPP
3  . . .
4  typedef union TTipo
5  {
6      uint32_t ValorInt;
7      float ValorFloat;
8      bool ValorLogico;
9  }TipoGenerico;
10 . . .
11
12 #endif //COMUNICACION TAREAS HPP
```

Código 4.130: Definición de la unión TipoGenerico, tipos de los datos de la carga útil del mensaje

Al tratarse de una unión, los elementos que contiene ocupan la misma posición de memoria, es decir, la unión solo contiene un único dato que puede ser accedido de diversas formas. Se definen tres tipos de datos: entero sin signo de 32 bits, flotante y valor lógico. Así, si por ejemplo se quiere mandar el mensaje de datos de espectro disponibles, desde la aplicación al entrono gráfico — *DATOS_ESPECTRO*—, que necesita mandar un puntero, dos datos en coma flotante y un dato entero, el código necesario sería:

```
1  uint8_t registro2[2048];
2  uint8_t *Datos_para_grafico=registro2;
3  Mensaje mensaje_emitido;
4  . . .
5  //En registro2 se calcula la FFT de datos adquiridos y se modifican los valores
6  //de posición y escala vertical.
7  . . .
8  //Se borra la especificación de tipo de evento del mensaje emitido anteriormente.
9  mensaje_emitido.evento&=~MASCARA_TIPO_EVENTO;
10 //El nuevo tipo de evento: nuevos datos FFT a presentar en pantalla
11 mensaje_emitido.evento |= DATOS_ESPECTRO;
12 //Se borra la especificación de tipo de subevento del mensaje emitido anteriormente.
13 mensaje_emitido.evento&=~MASCARA_TIPO_SUBEVENTO;
14 //Se especifica el subevento: además de datos FFT se ha modifucado la escala y posición
15 //verticales
16 mensaje_emitido.evento |= ESCALA_VER_DES_VER_MODIFICADOS;
17 //En carga1 se manda el valor del puntero (tipo entero) que apunto a registro2
18 mensaje_emitido.carga1.ValorInt = (uint32_t)(Datos_para_grafico);
19 //En carga2 se manda el nuevo valor de la posición vertical (tipo flotante)
20 mensaje_emitido.carga2.ValorFloat = (float)DesplazamientoVertical
21 //En carga3 se manda el nuevo valor de la escala vertical (tipo flotante)
22 mensaje_emitido.carga3.ValorFloat = (float)EscalaVertical;
23 //En carga4 se manda el canal en uso (tipo entero)
24 mensaje_emitido.carga4.ValorInt = (uint8_t) canal;
25 //Finalmente se manda el mensaje por la cola
26 xQueueSend(colaDeAplicacion_a_GUI, ( void * ) &mensaje_emitido, 0);
```

Código 4.131: Código ejemplo de creación y envío de mensaje entre contextos

En este código ejemplo se definen las variables *registro2*, que contendrá los datos a ser tratados por el contexto receptor del mensaje, el puntero *Datos_para_grafico*, que será cargado con la dirección de la variable anterior, y la variable *mensaje_emitido* de tipo la estructura *Mensaje*. La creación del mensaje consistirá en la carga de aquellos valores de la variable anterior, que sean necesarios para el mensaje concreto. En este caso, se pretende mandar el mensaje de tipo *DATOS_ESPECTRO* desde la aplicación hacia el modelo —que indica al modelo que hay datos que mostrar en pantalla—. El primer paso es indicar, en el campo evento de la variable que constituye el mensaje, que se trata del evento de «datos a mostrar en pantalla» — *DATOS_ESPECTRO* —. Para ello se ha de borrar de los bits que indican el tipo de mensaje, el contenido que tuvieran anteriormente —línea 9—. Luego se han de rellenar con la especificación de evento a mandar —línea 11—. Como en el ejemplo se necesita calificador del evento —subevento—, se han de borrar los bits correspondientes —línea 13—, para luego escribirlos con la información deseada —línea 16—. A continuación se ha de rellenar la carga útil del mensaje. Para cada dato de cada carga, se ha de especificar el tipo de dato que va a contener, tal y como se muestra en las líneas 18, 20, 22 y 24. Hay que señalar que para mandar un puntero se utilizará el especificador *ValorInt*, con la prudencia que en la recepción del mensaje se realice la conversión explícita al tipo de puntero correcto. Esta misma acción se debe llevar a cabo cuando lo que se pretenda mandar sea un dato de un tipo enumerado. El dato entero en la carga — *ValorInt* — también permite el envío de otros enteros distintos a 32 bits —menores— e incluso con signo. En este caso hay que realizar la conversión explícita al dato correcto tanto en la creación del mensaje como en la recepción. Un ejemplo de ello se encuentra en la línea 24, donde el dato de tipo *uint8_t* —*canal*— es cargado en el dato de tipo uint32_t previa conversión. Finalmente se procede a mandar el mensaje —*mensaje_emitido*— por la cola —*colaDeAlpicacion_a_GUI*—.

4.5.8 Método principal de la clase DispositivoFFT: tareaFFT.

El método *tareaFFT*, es el más importante de la clase *DispositivoFFT*. De hecho, el resto de elementos de la clase —atributos y métodos— son accesorios a él. Su función principal es la de ordenar y sincronizar la adquisición de los datos temporales —por parte de la configuración triple del ADC y el DMA—, convertir y procesar los datos, y realizar la comunicación con el entorno gráfico. El flujograma general que sigue esta función miembro es el siguiente:

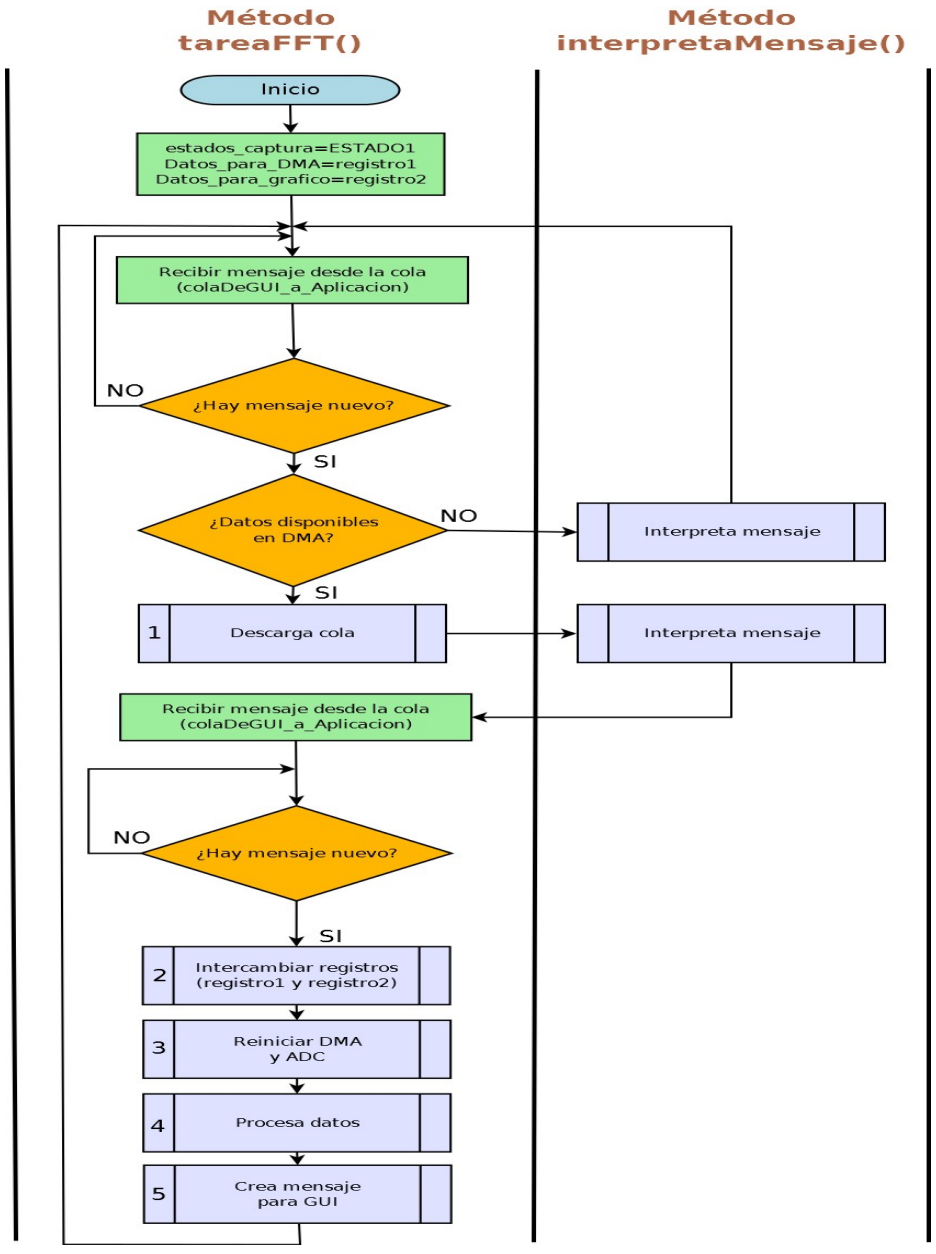


Ilustración 4.55: Flujograma general del método tareaFFT de la clase DispositivoFFT

Primeramente inicializa una serie de variables internas que son necesarias para realizar diversas acciones. Los valores de los atributos de la clase ya han sido determinados en la lista de inicialización del constructor o directamente en su cuerpo. Luego, entra en un bucle infinito del que no saldrá —a no ser que se apague o se reinicie el sistema—. Esto es debido a que el método *tareaFFT* estará encapsulado en una tarea del sistema operativo. Seguidamente, espera a recibir un mensaje por la cola del sistema, procedente del contexto gráfico. En

el caso de no haberse completado la adquisición, por parte del conjunto ADC, se procede a interpretar este mensaje, que consiste en determinar qué se solicita desde el entorno gráfico y qué datos se reciben. Esta acción se delega en otro de los métodos de la clase *DispositivoFFT*: el método *interpretaMensaje* —será descrito en la siguiente sección—. Si por el contrario, al recibir el mensaje, ya ha terminado la adquisición, se procede a descargar e interpretar los posibles mensajes que estén almacenados en la cola —como máximo 4, ya que ese es el tamaño de la misma— y esperar la llegada del siguiente mensaje. Esto se hace así para dejar la ejecución del método *tareaFFT* en disposición de mandar los datos adquiridos en el siguiente «tick» del entorno gráfico, asegurándose que se va a gestionar de forma inmediata el siguiente mensaje mandado por este, que en cualquier caso significará que se están solicitando datos. Una vez recibido este último mensaje, se realiza el intercambio de los dos registros utilizados en este método, uno para la adquisición de datos y otro para ser utilizado por el entorno gráfico —constituyendo la memoria compartida entre contextos, ver sección 4.5.7—. Luego, se reinicia el funcionamiento del ADC y DMA —el DMA ha sido deshabilitado desde su rutina de interrupción, cuando se ha transferido la cantidad de datos preestablecidos (ver sección 4.5.5); el reinicio del ADC asegura que se adquieran los datos justo después del reinicio del DMA—. A continuación, se procesan los datos adquiridos para ser entregados. Finalmente, se elabora el mensaje que será enviado al contexto gráfico. El código que implementa este flujograma es:

```
1  void DispositivoFFT::tareaFFT(void)
2  {
3    static float Maxi=-1e6;
4    static float Mini=1e6;
5    static int i;
6
7    uint8_t *Datos_para_grafico=registro2;
8    uint8_t *Datos_para_DMA=registro1;
9    uint16_t dato;
10
11
12   while (1)
13   {
14       if (xQueueReceive(colaDeGUI_a_Aplicacion, &mensaje_recibido, portMAX_DELAY))
15       {
16           if (xSemaphoreTake(semaf_datos_disponibles, 0)==pdFALSE)
17               interpretaMensaje();
18           else
19           {
20             /*******************************/
21             /*** PROCESO 1: Descarga cola ***/
22             /*******************************/
23
24              if (xQueueReceive(colaDeGUI_a_Aplicacion, &mensaje_recibido, portMAX_DELAY ))
25             {
26
27               /*****************************************/
28               /*** PROCESO 2: Intercambia registros  ***/
29               /*****************************************/
30
31               /*************************************/
32               /*** PROCESO 3: Reinicia DMA y ADC ***/
33               /*************************************/
34
35               /*******************************/
36               /*** PROCESO 4: Procesa Datos ***/
37               /*******************************/
38
39               /***************************************/
40               /*** PROCESO 5: Creación del mensaje ***/
41               /***************************************/
42
43               }
44           }
45       }
46   }
47
48 }
```

Código 4.132: Implementación del flujograma general del método tareaFFT de la clase DispositivoFFT

La primera espera del mensaje se realiza con el bloqueo de la tarea de forma indefinida aguardando el mensaje proveniente desde la cola *colaDEGui_a_Aplicacion* —línea 14—. El mensaje recibido es almacenado en

el atributo *mensaje_recibido* de la clase para que luego sea utilizado por el método *interpretaMensaje*. Si hay no datos disponibles desde el DMA —testigo dado en su rutina de atención a la interrupción, ver sección 4.5.5—, se procede a interpretar el mensaje, pero sin bloquear la tarea —líneas 16 y 17—. Al coger el testigo en este semáforo —línea 16— el testigo desaparece y se necesita que la interrupción envíe uno nuevo al finalizar una nueva transferencia de datos —DMA—. Si hay datos disponibles al recibir el mensaje, no solo se interpreta este, sino que se interpretan todos los posibles que pueda contener la cola —PROCESO 1: Descarga cola— para a continuación bloquearse hasta el siguiente mensaje —línea 24— que solicitará datos nuevos. Cuando se recibe este mensaje, se intercambian los registros —PROCESO 2: Intercambia registros—, se reinicia la captura —PROCESO 3: Reinicia DMA y ADC—, se procesan los datos capturados —PROCESO 4: Procesa Datos— y se envía un mansaje al contexto gráfico con los datos procesados —PROCESO 5: Creación del mensaje—. Cada uno de estos procesos constituye una porción de código del método *tareaFFT* con una funcionalidad concreta. En el caso de PROCESO 1, su diagrama de flujo es:

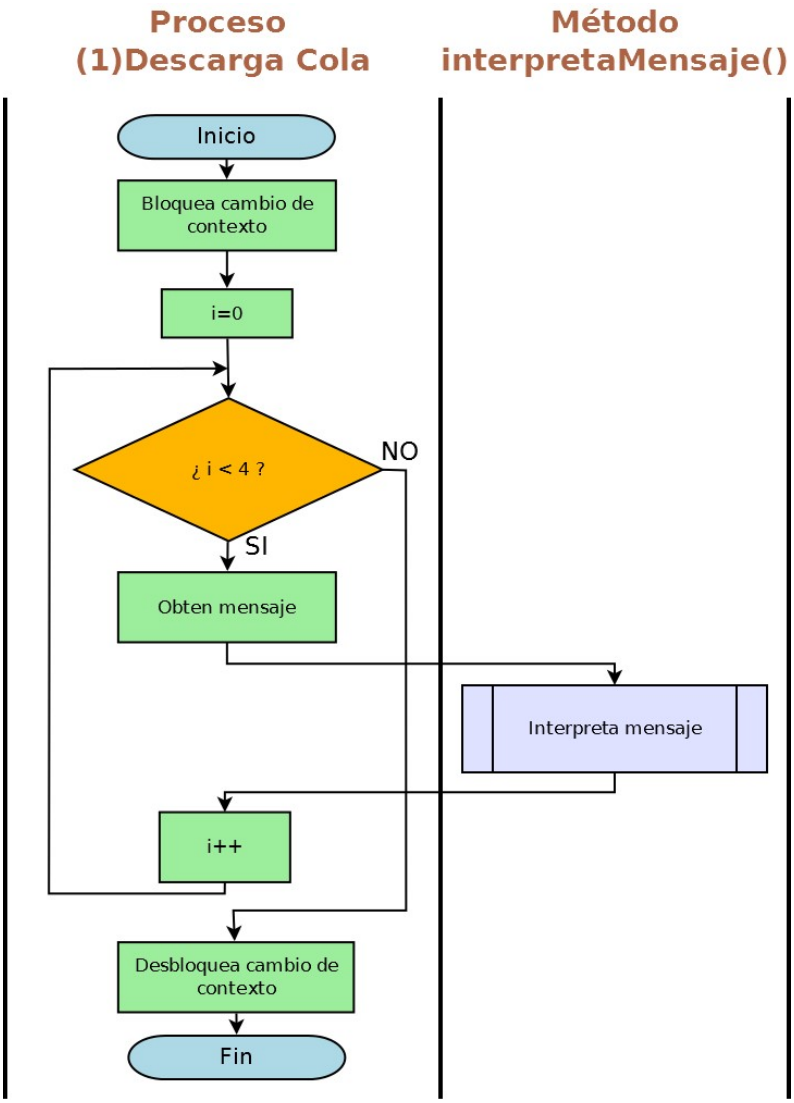


Ilustración 4.56: Flujograma del proceso «(1)Descarga Cola» del método tareaFFT

Se trata realmente de bloque tipo *for*, donde se va sacando el último dato de la cola —en total hasta 4— e interpretándolo. Su código de implementación es:

```
1           . . .
2
3           /*******************************/
4           /*** PROCESO 1: Descarga cola ***/
5           /*******************************/
6           taskENTER_CRITICAL();
7           for (i=0; i<4; i++)
8           {
9             xQueueReceive(colaDeGUI_a_Aplicacion, &mensaje_recibido, ( portTickType  )0);
10            interpretaMensaje();
11          }
12          taskEXIT_CRITICAL();
13
14          . . .
```

Código 4.133: Implementación del flujograma del proceso «(1)Descarga Cola» del método tareaFFT

La descarga de la cola se realiza entre las instrucciones taskENTER_CRITICAL y taskEXIT_CRITICAL del sistema operativo, para asegurar que otra tarea no se ejecute mientras se realice esta descarga —lo que asegura que el contexto gráfico no introduzca un nuevo mensaje desde el otro extremo de la cola—. El bucle se ejecuta 4 veces —tamaño de la cola— y la extracción del dato de la misma se realiza sin bloqueo o tiempo de espera, ya que puede darse la situación de no estar totalmente llena e incluso estar totalmente vacía. Cada mensaje extraído es interpretado mediante la llamada al método *interpretaMensaje*.El flujograma del PROCESO 2 es:

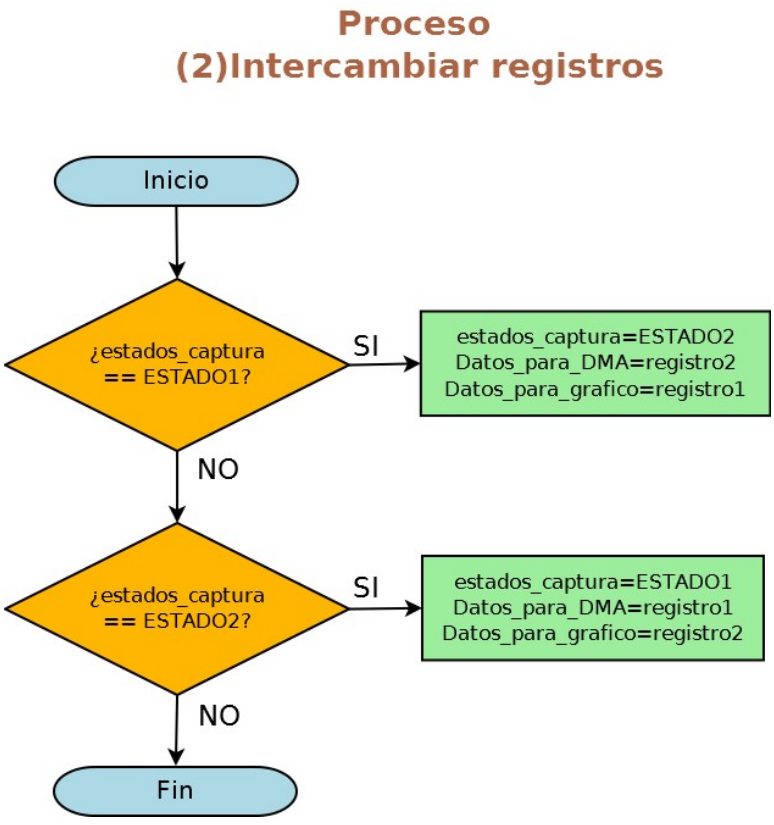


Ilustración 4.57: Flujograma del proceso «(2)Intercambiar registros» del método tareaFFT

Realmente se trata de una máquina de estados —con los estados *ESTADO1* (por defecto) y *ESTADO2*— pero, debido a que todo el funcionamiento se está describiendo mediante diagramas de flujo, es la forma conveniente de hacerlo. En cada estado se hace que el siguiente a contemplar sea el otro estado —para que la máquina de estados esté alternando entre ambos—, y en cada uno de ellos se intercambien los punteros que apuntan al registro que usará el DMA —puntero Datos_para_DMA— y el que se usará para el entrono gráfico —*Datos_para_grafico*—. Para implementar este flujo se utiliza el siguiente código:

```
1           . . .
2
3           /****************************************/
4           /*** PROCESO 2: Intercambia registros  ***/
5           /****************************************/
6           switch (estados_captura)
7            {
8             case ESTADO1:
9                  Datos_para_grafico=registro1;
10                 Datos_para_DMA=registro2;
11                 estados_captura = ESTADO2;
12                 break;
13            case ESTADO2:
14                 Datos_para_grafico=registro2;
15                 Datos_para_DMA=registro1;
16                 estados_captura = ESTADO1;
17                 break;
18           }
19          . . .
```

Código 4.134: Código del flujograma del proceso «(2)Intercambia registros» del método tareaFFT

Para la máquina de estados se utiliza el atributo *estados_captura* de la clase *DispositivoFFT* de tipo *EstadosCaptura* —ver sección 4.5.4— cuyos valores posibles son *ESTADO1* y *ESTADO2*. En el constructor de la clase, este atributo es puesto a *ESTADO1*, y los punteros *Datos_para grafico* y *Datos_para_DMA* a los valores respectivos de registro2 y registro1, en la zona de inicialización del método *tareaFFT*. En cada entrada en el bloque *switch-case*, se especifica que el siguiente estado es el estado contrario y se procede a intercambiar el contenido apuntado por los punteros. De esta forma, cuando en otra parte del código se haga referencia a *Datos_para_grafico*, siempre se tratará del array que va a manejar el entorno gráfico, independientemente si en ese momento el puntero está cargado con la dirección de *registro1* o *registro2*. Igual ocurre con *Datos_para_DMA*, que siempre apunta a los datos que va a utilizar el DMA como destino de la transferencia de datos provenientes de la captura en el ADC. El PROCESO 3 sigue el siguiente diagrama de flujo:

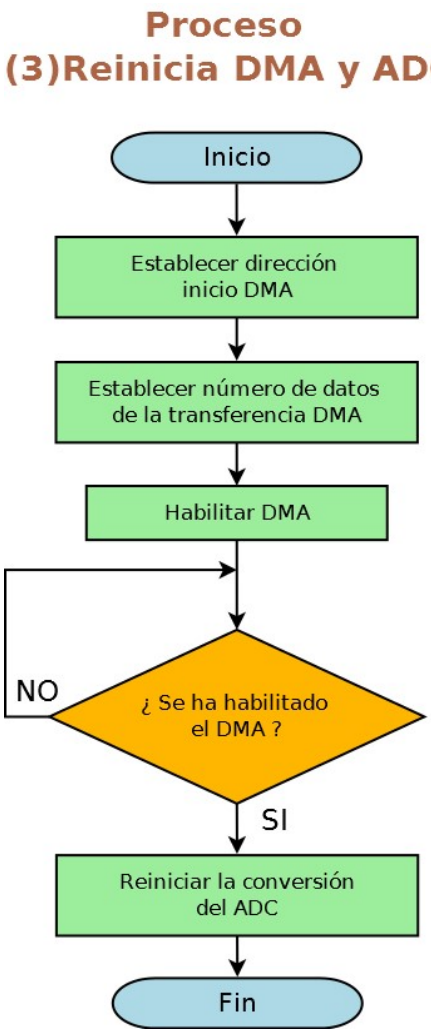


Ilustración 4.58: Flujograma del proceso «(3)Reinicia DMA y ADC» del método tareaFFT

Se trata de volver a habilitar la transferencia del DMA, primero se especifica hacia donde se transferirán los datos, y el número de ellos a transferir. Luego se ordena la habilitación y se espera hasta que realmente se ha producido, luego se ordena al ADC a reiniciar la captura. El código que lo implementa es:

```
1           . . .
2
3           /************************************/
4           /*** PROCESO 3: Reinicia DMA y ADC ***/
5           /************************************/
6           DMA2_Stream0->M0AR=(uint32_t)Datos_para_DMA;
7           DMA_SetCurrDataCounter(DMA2_Stream0, 1024);
8           DMA_Cmd(DMA2_Stream0, ENABLE);
9           while (DMA_GetCmdStatus(DMA2_Stream0)!=ENABLE);
10          ADC_SoftwareStartConv(ADC1);
11
12          . . .
```

Código 4.135: Código del flujograma del proceso «(3)Reinicia DMA y ADC» del método tareaFFT

No existe una función en la librería de periféricos estándar de micro STM32F4 donde se recargue la dirección de inicio de la transferencia DMA, por lo que se utilizan los punteros y valores preestablecido en el archivo *stm32f4xx.h* —línea 6—. En esta línea se aprecia que se hace referencia al puntero que apunta a los datos a ser utilizados para la transferencia, independientemente si son *registro1* o *registro2*. EL número de datos a transferir

es de 2048 bytes —lo que equivale a 1024 datos de longitud 16 bits, ver sección 4.5.5—, especificado en la línea 7. La habilitación y la espera para que ello suceda, se realizan en las líneas 8 y 9 respectivamente. Finalmente se reinicia la captura del ADC en la línea 10.

El PROCESO 4 es el más complejo de todos, su misión es el tratamiento de los datos capturados. Debido a esta complejidad, su flujograma se divide en bloques funcionales, tal y como se muestra en la siguiente figura:

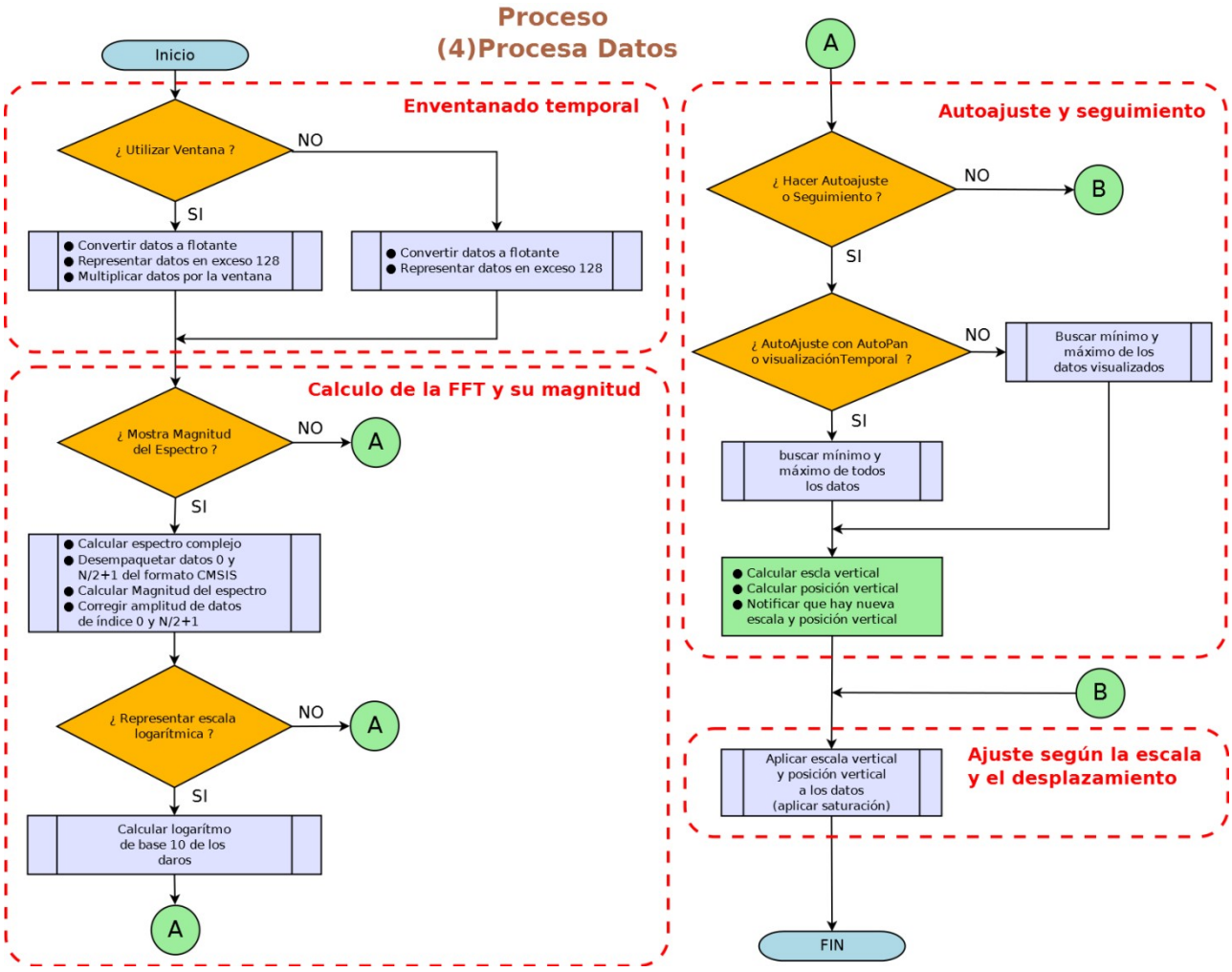


Ilustración 4.59: Flujograma del proceso «(4)Procesa Datos» del método tareaFFT

En el *enventanado temporal*, se decide si se aplica o no, la función de ventana a los datos capturados —ver secciones 4.5.3 y 4.5.6—. En cualquiera de los casos, los datos son convertidos a formato de coma flotante —necesario para utilizar el algoritmo FFT de CMSIS, ver sección 4.5.2— y con exceso 128 —el dato 128 representa el cero, el 255 el 127 y el 0 el -128, con lo que la entrada al ADC, que va de 0V a 3V, ha de ser montada sobre 1,5V—. El código que implementa este bloque es:

```
1
2           /*******************************/
3           /*** PROCESO 4: Procesa Datos ***/
4           /*******************************/
5
6            /*** Enventanado o no de los datos temporales ***/
7
8           if (Ventana)
9            {
10            for (i=0; i<N_DATOS_USADOS; i++)
11                flotante[i]=(((float)(Datos_para_grafico[i])-128.0f)*ventana[i]);
12           }
13          else
14           {
15            for (i=0; i<N_DATOS_USADOS; i++)
16                flotante[i]=(((float)(Datos_para_grafico[i])-128.0f));
17           }
18
19           . . .
```

Código 4.136: Enventanado del flujograma del proceso «(4)Procesa Datos» del método tareaFFT

Primeramente se verifica si se ha de aplicar la ventana —atributo booleano *Ventana* de la clase, en línea 8—. Esta variable miembro ha sido rellenada por el método *ponVentana* —ver sección 4.5.6— llamado por *interpretaMensaje* al interpretar el mensaje. Si se ha de aplicar una ventana, los datos capturados por el ADC son multiplicados por la función de ventana almacenada en el array *ventana* —línea 11—, array que ha sido rellenado por el método *ponVentana* igualmente. En cualquier caso, los datos son convertidos a flotante y se les aplica el exceso 128 —líneas 11 y 16—.

En el siguiente bloque funcional se realiza el cálculo de la FFT, su magnitud y el escalado logarítmico. El código que lo implementa es:

```
1           /*******************************/
2           /*** PROCESO 4: Procesa Datos ***/
3           /*******************************/
4
5            . . .
6
7            /*** Calculo de la FFT y su magnitud ***/
8
9            if (!temporal)
10            {
11             arm_rfft_fast_f32(&S,  flotante, flotante2, 0);
12             flotante2[2048]=flotante2[1];
13             flotante2[2049]=0;
14             flotante2[1]=0;
15             arm_cmplx_mag_f32(flotante2, flotante, N_DATOS_USADOS);
16             /*** Corrección del nivel de continua para el espectro unilateral ***/
17             flotante[0]/=2;
18             flotante[1024]/=2;
19             /*** Escalado logarítmico de los datos frecuenciales ***/
20             if (EscalaLogaritmica)
21              {
22               for (i=0; i<N_DATOS_USADOS/2+1; i++)
23                {
24                  if (flotante[i]==0)flotante[i]=1e-10;//evita calcular el log(0)
25                  flotante[i]=log10(flotante[i]);
26                }
27              }
28            }
29
30            . . .
```

Código 4.137: Magnitud de la FFT del flujograma del proceso «(4)Procesa Datos» del método tareaFFT

Solo se procede al escalado logarítmico —código entre la líneas 20 a 27— si lo que se quiere mostrar es la magnitud del espectro —discriminado en la línea 9—. Este discernimiento se hace mediante la variable miembro *temporal*, variable que es actualizada al interpretar el mensaje. Para mostrar la magnitud del espectro, primero se calcula el espectro complejo de los datos reales de entrada —línea 11, cálculo de la rfft directa, ver sección

4.5.2—. Luego se desempaquetan los datos complejos de índice 0 y 1024, que al solo tener parte real —parte imaginaria igual a cero—, CMSIS los empaqueta en el primer par complejo de tal forma que en la parte real de este par, está la parte real del dato de índice cero —nivel de continua—, mientras que en su parte imaginaria está la parte real del dato 1024. Hay que tener en cuenta que CMSIS no maneja datos en formato complejo, por ello el primer par complejo —el complejo de índice cero— ocupan los lugares cero y uno del array flotante2 —datos real e imaginario, respectivamente— mientras que el complejo 1024 ocupan las posiciones 2048 y 2049. Este desempaquetado es realizado entre las líneas 12 a 14. Luego, en la línea 15, se calcula la magnitud del espectro, almacenando el resultado en el array *flotante*. En las líneas 17 y 18 se realiza la corrección de nivel que es diferente para los datos de índice cero —nivel de continua— y 1024 —frecuencia de Nyquist—, la corrección común de nivel para todos los datos es delegada en el contexto gráfico —para más información de la corrección de nivel, ver sección 4.5.3—. Finalmente, si se decide mostrar los datos en escala logarítmica, se procede a ello asegurándose de no realizar el logaritmo de cero.

En el siguiente bloque funcional del PROCESO 4, se realiza el autoajuste de los datos y seguimiento mediante el siguiente código:

```
1           /*******************************/
2           /*** PROCESO 4: Procesa Datos ***/
3           /*******************************/
4
5            . . .
6
7           /*** Autoajuste y seguimiento ***/
8
9            if (Seguimiento || !AutoAjusteHecho)
10            {
11              Maxi=-1e6;
12              Mini=1e6;
13              if (!AutoPan || temporal)
14               {
15                 for (i=0; i<N_DATOS_USADOS/2+1; i++)
16                   {
17                     if (((float)(flotante[i]))>Maxi) Maxi=flotante[i];
18                     if (((float)(flotante[i]))<Mini) Mini=flotante[i];
19                   }
20               }
21              else
22               {
23                 for (i=IndiceInicial; i<IndiceFinal; i++)
24                   {
25                     if (((float)(flotante[i]))>Maxi) Maxi=flotante[i];
26                     if (((float)(flotante[i]))<Mini) Mini=flotante[i];
27                   }
28               }
29              AutoAjusteHecho=true;
30              EscalaVertical=255.0f/(Maxi-Mini);
31              DesplazamientoVertical=Mini*EscalaVertical;
32              escalaVerPosVerModificada=true;
33            }
34
35           . . .
```

Código 4.138: Autoajuste y seguimiento del flujograma del proceso «(4)Procesa Datos» del método tareaFFT»

El autoajuste consiste en ajustar el máximo y mínimo de los datos tratados a los límites verticales de la pantalla de forma puntual, mientras que el seguimiento lo hace de forma permanente en cada presentación. Para ambas funcionalidades, la clase *DispositivoFFT* dispone de los atributos *AutoAjusteHecho* y *Seguimiento*. Para realizar el autoajuste, el atributo *AutoAjusteHecho* se debe poner momentáneamente a falso; cuando el autoajuste es realizado automáticamente vuelve al valor verdadero. En el caso del seguimiento, hay que poner explícitamente la variable miembro Seguimiento a verdadero para realizarlo, y volver a ponerla explícitamente a falso para no realizar seguimiento. El seguimiento y autoajuste se puede hacer teniendo en cuenta todo el registro —*AutoPan* a falso— o solo los datos visualizados —*Autopan* a verdadero—. Cuando se realiza el autoajuste o seguimiento se deben calcular el máximo y mínimo de los datos. En el caso de *AutoPan* a falso, este máximo y mínimo son determinados teniendo en cuenta todo el registro —líneas de la 13 a la 20—, mientras que

si es falso, solo se tienen en cuenta los datos visualizados —líneas de la 21 a la 28—. Si lo que se visualiza es la representación temporal de los datos, se tiene en cuenta todo el registro para el cálculo del máximo y mínimo —líneas de la 13 a la 20—. Los índices de los datos visualizados —IndiceInicial e IndiceFinal en la línea 23— vienen como datos en el mensaje entrante —ver siguiente sección—. En cualquiera de los casos, siempre se pone la variable miembro *AutoAjusteHecho* a verdadero —línea 29—, para que la siguiente iteración de la *tareaFFT* no se vuelva a realizar el autoajuste si no es explícitamente indicado. También se determina la escala vertical —almacenada en la variable miembro *EscalaVertical*— que constituye el factor por el que multiplicar los datos para ajustarse a la altura de pantalla —línea 30—, así como el desplazamiento —valor que restar a los datos escalados para que todos estén dentro de la pantalla, realizado en la línea 31—. Finalmente se pone la variable miembro *escalVerPosVerModificado* a verdadero para indicar al PROCESO 5 —el encargado de realizar y mandar el mensaje de salida hacia el contexto gráfico— que tenga en cuenta que tanto la escala como el desplazamiento vertical han sido modificados a la hora de realizar el mensaje de salida. Hay que aclarar que cuando se realiza este ajuste de datos, se hace para que la diferencia máxima entre el valor mayor y menor sea de 255, ya que este es el valor máximo que puede representar un entero sin signo de 8 bits —debido a que los datos serán mandados al contexto gráfico como un array de bytes—. Pero 255 no es el valor de altura de la gráfica en pantalla. Se necesita una corrección de visualización que se realiza en el contexto gráfico, concretamente en el objeto que representa la gráfica en la vista de *TouchGFX.* La gráfica va a tener una altura de dos tercios de la pantalla utilizada. Ya que la pantalla utilizada tiene una altura de 272 puntos, la altura de la gráfica será de 181 puntos, por tanto, los datos recibidos deben ser multiplicado por un factor de 0.7098 —se redondea a 0,7— en el objeto de tipo Serie del gráfico —ver sección 4.4.1.11— donde utilizará el atributo *EscalaVert* —a través de su método de acceso *ponEscalaVert*— para albergar este factor.

El último bloque funcional del PROCESO 4 es el encargado de aplicar la escala y desplazamiento del bloque anterior. Su código es el siguiente:

```
1           /*********************************/
2           /*** PROCESO 4: Procesa Datos ***/
3           /*********************************/
4
5            . . .
6
7           /*** Ajuste según la escala y el desplazamiento ***/
8
9           for (i=0; i<N_DATOS_USADOS; i++)
10           {
11             dato = flotante[i]*EscalaVertical-DesplazamientoVertical;
12             if (dato>255)
13                 Datos_para_grafico[i]=255;//saturación
14             else
15                 Datos_para_grafico[i]=(uint8_t)dato;
16           }
```

Código 4.139: Escala y posición vertical del flujograma del proceso «(4)Procesa Datos» del método tareaFFT

Se recorren todos los datos aplicándoles la escala y desplazamiento vertical, antes calculados, saturando aquellos datos que superan el valor de 255. Estos datos corregidos son almacenados en el array apuntado por el puntero Datos_para_grafico que es de tipo puntero a uint8_t, por lo que los datos son convertidos implícitamente a enteros de 8 bits sin signo.

El último proceso del método tareaFFT es el PROCESO 5 dedicado a la creación y envío del mensaje con los datos procesados. Para ello se deben borrar aquellos campos del mensaje de salida hacia el contexto gráfico que identifican el mismo para a continuación rellenarlos con el nuevo identificador, luego, si se ha modificado la escala y desplazamiento vertical, indicarlo como calificador del evento y finalmente rellenar la carga útil del mensaje —ver sección 4.5.7—. El flujograma de este proceso es el siguiente:

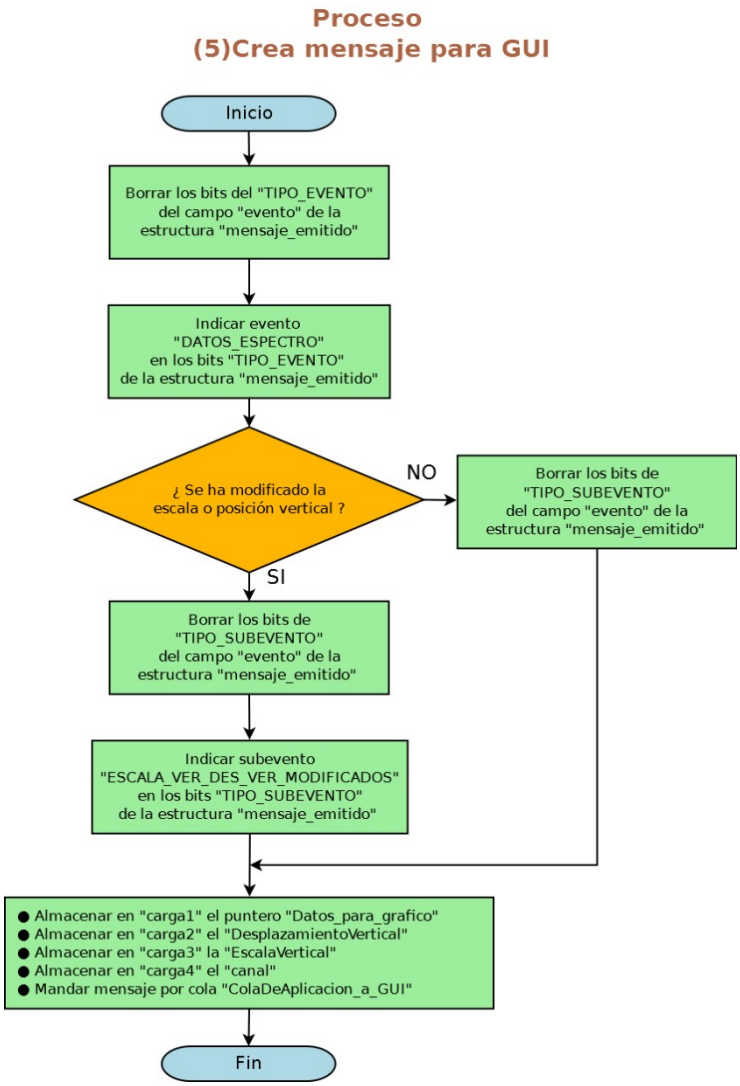


Ilustración 4.60: Flujograma del proceso «(5)Crea mensaje para GUI» del método tareaFFT

Y el código que lo implementa es:

```
        /*************************************/
        /*** PROCESO 5: Creación del mensaje ***/
        /*************************************/
        mensaje_emitido.evento &=~ MASCARA_TIPO_EVENTO;
        mensaje_emitido.evento |= DATOS_ESPECTRO;

        if (escalaVerPosVerModificada)
        {
          mensaje_emitido.evento&=~MASCARA_TIPO_SUBEVENTO;
          mensaje_emitido.evento |= ESCALA_VER_DES_VER_MODIFICADOS;
          escalaVerPosVerModificada=false;
        }
        else
          mensaje_emitido.evento&=~MASCARA_TIPO_SUBEVENTO;

        mensaje_emitido.carga1.ValorInt = (uint32_t)(Datos_para_grafico);
        mensaje_emitido.carga2.ValorFloat = (float)DesplazamientoVertical;
        mensaje_emitido.carga3.ValorFloat = (float)EscalaVertical;
        mensaje_emitido.carga4.ValorInt = (uint8_t) canal;
        xQueueSend(colaDeAplicacion_a_GUI, ( void * ) &mensaje_emitido, 0);
```

Código 4.140: Código del flujograma del proceso «(5)Crea mensaje para GUI» del método tareaFFT

El mensaje es creado en el atributo *mensaje_emitido*. Primero se especifica el identificador del mensaje —línea 4 y 5—. Luego, si se han modificado la escala y desplazamiento vertical, se identifica el subevento –línea 9 y 10— y se restituye el valor por defecto del atributo *escalaVerPosVerModificada* —línea 11, para que en el siguiente mensaje no se informe del cambio de la escala o desplazamiento verticales si no es hecho explícitamente—. Si no hay modificación de tales valores, se borra la información del subevento que se utilizó en el envío de mensaje anterior —línea 14—. A continuación se rellenan la carga útil —líneas de a 16 a la 19— consistente en un primer dato, en el que se especifica la dirección de la región de memoria donde están los datos a representar; un segundo y tercer dato, consistente en el desplazamiento y la escala vertical, actualmente en uso —independientemente si han sido modificados o no—; y un cuarto dato que especifica el canal que utiliza el ADC —cuyos valores posibles son 1 y 2—. Finalmente se manda el mensaje por la cola de salida hacia el entorno gráfico —*colaDeAplicacion_a_GUI*— sin bloquear la tarea ni esperar ningún tiempo —línea 20—.

4.5.9 Intérprete de mensajes entrantes: método «interpretaMensaje»

El método *interpretaMensaje* de la clase *DispositivoFFT* es el encargado de analizar el contenido de los mensajes que provienen del contexto gráfico y actuar en consecuencia. La recepción del mensaje es llevada a cabo por la tarea principal de la clase, *tareaFFT*, que guarda el mensaje entrante en el atributo *mensaje_recibido*. La tarea principal delega la interpretación en este método que recoge el contenido de *mensaje_recibido* y modifica los atributos pertinentes de la clase *DispositivoFFT* para que cuando se retorne a *tareaFFT*, esta tenga en cuenta estos cambios. El esquema de colaboración entre estos dos métodos es el siguiente:

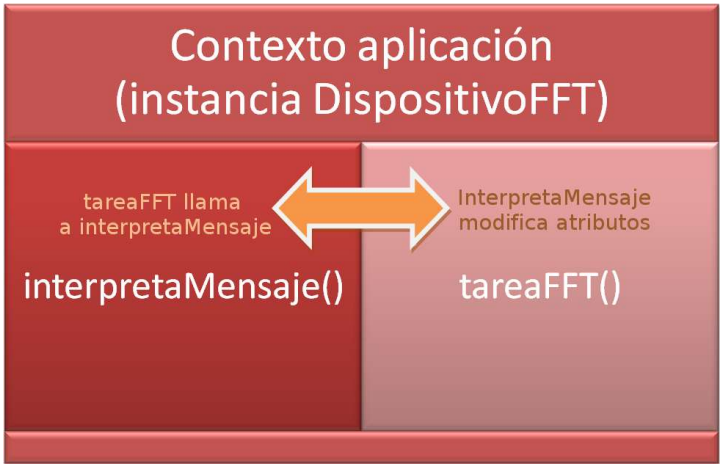


Ilustración 4.61: Esquema de cooperación entre los métodos tareaFFT e interpretaMensaje de la clase DispositivoFFT

La mayor parte de los tipos de mensajes son enviados por el contexto gráfico a la aplicación. Los tipos de mensajes que tiene que gestionar la aplicación —instancia de la clase *DispositivoFFT*— son los resaltados en la siguiente ilustración:

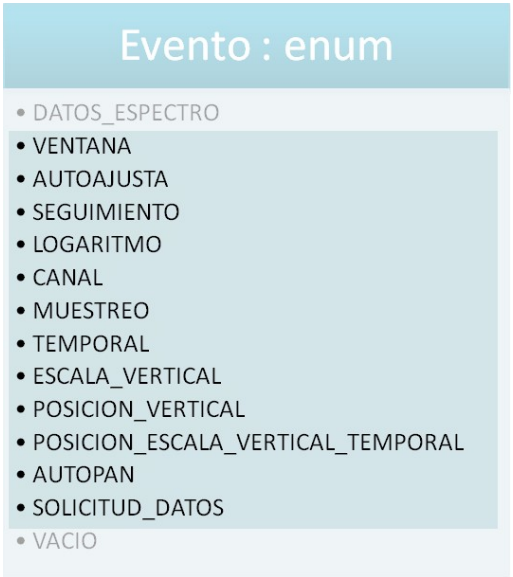


Ilustración 4.62: Tipos de mensajes a ser gestionados por el método interpretaMensaje

El tipo de mensaje *DATOS_ESPECTRO* es enviado por la *tareaFFT* hacia el entorno gráfico y ha sido tratado en la sección 4.5.7. El identificado del mensaje *VACIO* es cero, y está reservado para posibles usos futuros. El resto de tipos de mensajes —los resaltados—, han de ser gestionados por el método *interpretaMensaje*. Su código es el siguiente:

```
1  void DispositivoFFT::interpretaMensaje(void)
2  {
3      mensaje_emitido.evento&=~MASCARA_DATO_EVENTO
4      switch ((mensaje_recibido.evento & MASCARA_TIPO_EVENTO))
5      {
6          case VENTANA:
7              ponVentana((TiposVentana)mensaje_recibido.carga1.ValorInt);
8              mensaje_emitido.evento|=DATO_RECIBIDO;
9              break;
10         case AUTOAJUSTA:
11             autoAjustar();
12             IndiceInicial=mensaje_recibido.carga1.ValorInt;
13             IndiceFinal=mensaje_recibido.carga2.ValorInt;
14             mensaje_emitido.evento|=DATO_RECIBIDO;
15             break;
16         case SEGUIMIENTO:
17             seguimientoActivo(mensaje_recibido.carga1.ValorLogico);
18             mensaje_emitido.evento|=DATO_RECIBIDO;
19             break;
20         case LOGARITMO:
21             escalarLogaritmico(mensaje_recibido.carga1.ValorLogico);
22             autoAjustar();
23             mensaje_emitido.evento|=DATO_RECIBIDO;
24             break;
25         case CANAL:
26             ponCanal(mensaje_recibido.carga1.ValorInt);
27             mensaje_emitido.evento|=DATO_RECIBIDO;
28             break;
29         case MUESTREO:
30             ponFrecMuestreador((RelojMuestreador)mensaje_recibido.carga1.ValorInt);
31             mensaje_emitido.evento|=DATO_RECIBIDO;
32             break;
33         case TEMPORAL:
34             verTemporal(mensaje_recibido.carga1.ValorLogico);
35              if (temporal==true)autoAjustar();
36             mensaje_emitido.evento|=DATO_RECIBIDO;
37             break;
38         case POSICION_VERTICAL:
39             ponerPosicionVertical(mensaje_recibido.carga1.ValorFloat);
40             mensaje_emitido.evento|=DATO_RECIBIDO;
41             break;
42         case ESCALA_VERTICAL:
43             ponerEscalaVertical(mensaje_recibido.carga1.ValorFloat);
44             mensaje_emitido.evento|=DATO_RECIBIDO;
45             break;
46         case POSICION_ESCALA_VERTICAL_TEMPORAL:
47             temporal=mensaje_recibido.carga3.ValorLogico;
48             EscalaVertical=mensaje_recibido.carga1.ValorFloat;
49             DesplazamientoVertical=mensaje_recibido.carga2.ValorFloat;
50             escalaVerPosVerModificada=true;
51             mensaje_emitido.evento|=DATO_RECIBIDO;
52             break;
53         case AUTOPAN:
54             AutoPan=mensaje_recibido.carga1.ValorLogico;
55             mensaje_emitido.evento|=DATO_RECIBIDO;
56             break;
57         case SOLICITUD_DATOS:
58             IndiceInicial=mensaje_recibido.carga1.ValorInt
59             IndiceFinal=mensaje_recibido.carga2.ValorInt
60             break;
61         default:
62             ;
63             break;
64     }
65 }
```

Código 4.141: Código del método interpretaMensaje de la clase DispositivoFFT

Se trata de un método sin argumentos, ya que el contenido del mensaje recibido ha sido almacenado en un atributo de la clase *DispositivoFFT* —el atributo es *mensaje_recibido*— por parte de *tareaFFT*. A pesar de tener que interpretar el mensaje recibido, la primera acción a realizar es dejar el mensaje de salida —el que *tareaFFT* elabora con los datos a representar en el contexto gráfico y que almacena en el atributo *mensaje_emitido*— en disposición de poder concretar los bits del campo evento que están dedicados al dato encapsulado dentro de este —borrando estos bits de este campo, realizado en la línea 3—. Esto es debido a que tras interpretar los distintos tipos de mensajes, el método *interpretaMensaje* indica, en los bits del dato del campo evento del mensaje de salida —*mensaje_emitido*—, que el mensaje ha sido recibido correctamente —líneas 8, 14, 18, 23, 27

31, 36, 40, 44, 51 y 55— para que el modelo reinicie el tipo del último mensaje enviado y en el siguiente mensaje que emita —en el siguiente tick— sea del tipo *SOLICITUD_DATOS* —a no ser que el modelo tenga otra orden que mandar—. De hecho, la única interpretación de un mensaje que no indica al modelo que se ha recibido el mensaje, es cuando *interpretaMensaje* recibe *SOLICITUD_DATOS*, ya que ello indica que el modelo ya tiene como último indicador de mensaje *SOLICITUD_DATOS*, y si no hay ninguna otra orden por parte del modelo, este será el tipo de mensaje que emita en el siguiente mensaje. Este mecanismo permite dos cosas: que el mensaje emitido por el modelo se repita hasta que la aplicación lo haya recibido, y que una vez recibido, por parte de la aplicación, el modelo no vuelva a enviar la última orden y la que envíe sea la solicitud de datos.

El método *interpreteMensaje* consiste en un bloque switch-case, donde se evalúa el tipo de mensaje recibido —línea 4— y se interpreta —cada case-break en las líneas de la 6 a la 63—.

Con el mensaje de tipo *VENTANA*, el modelo solicita a la aplicación —instancia de la clase *DispositivoFFT*— que aplique la ventana que viene definida en la *carga1* a los datos adquiridos. Para ello, *interpretaMensaje* llama al método *ponVentana*, pasándole como argumento, previa conversión explícita a *TiposVentana* —ver sección 4.5.4—, el valor entero de *carga1*. El método *ponVentana*, creará la función de ventana solicitada —mediante la llamada a los métodos *crea_hanning*, *crea_hamming*, *crea_blackman_harris* o *crea_bartlett*, ver secciones 4.5.3 y 4.5.6— en el array *ventana* —definido en el archivo DatosEntreTareas.cpp, ver sección 4.5.7— y pondrá a valor verdadero el atributo booleano Ventana para que el método *tareaFFT* sepa que tiene que aplicar una función de ventana.

El mensaje *AUTOAJUSTA*, obliga a la *tareaFFT* a ajustar los datos al rango de 0 a 255 —ver sección 4.5.8—. Ello lo consigue mediante la puesta del atributo *AutoajusteHecho* a falso a través de la llamada al método de acceso *autoAjustar* —línea 11—. Esto hace que, mediante la inspección de este atributo, *tareaFFT* realice el ajuste. Una vez hecho dicho ajuste, el propio método *tareaFFT* vuelve a poner el atributo *AutoAjusteHecho* a verdadero para que en la próxima iteración no se realice un autoajuste si no se ha ordenado explícitamente. En el mensaje también viene la información de los índices incial y final de los datos visualizados en *carga1* y *carga2* respectivamente —líneas 12 y 13—. Esta última información es necesaria ya que si se ha activado la funcionalidad de *AutoPan*, solo se debe hacer el autoajuste en los datos visualizados —aquellos comprendidos entre *IndiceInicial* e *IndiceFinal*—.

El mensaje *SEGUIMIENTO* es similar al *AUTOAJUSTE*. La diferencia es que el primero mantiene el autoajuste de forma permanente en cada iteración de *tareaFFT*. Ello se consigue poniendo el atributo *Seguimiento* a verdadero a través de la llamada al método de acceso *seguimientoActivo*, cuyo argumento es el valor lógico de *carga1* que contiene el mensaje para indicar si se activa el seguimiento —valor verdadero— o si se desactiva el mismo —valor falso—.

El mensaje *LOGARITMO*, instruye a la aplicación a emplear el logaritmo de base 10 sobre la magnitud del espectro calculado. Para ello llama al método de acceso *escalaLogaritmico*, pasándole como argumento el valor lógico de la *carga1* del mensaje, que representa si se aplica la escala logarítmica —valor verdadero de *carga1*— o no —valor falso—. El método *escalaLogaritmico* se limita a poner el valor que recibe como argumento en el atributo *EscalaLogaritmica*, que es el dato que usara *tareaFFT* para aplicar o no la escala logarítmica. Una vez hecho esto se procede al autoajuste de los datos mediante la llamada al método autoAjustar.

Para indicar el cambio de canal desde el entorno gráfico a la aplicación, el primero manda a la segunda el mensaje de tipo *CANAL*, conteniendo en *carga1* los posibles valores enteros de los canales, es decir, 1 o 2. Para ello, el método *interpretaMensaje* llama al método *ponCanal*, pasándole como argumento *carga1*, que modifica el atributo *canal* acorde con el valor del argumento pasado. De acuerdo con el valor del atributo *canal*, tareaFFT seleccionará el canal 1 —canal 12 de la configuración triple entrelazada del ADC del micro STM32F4, ver sección 4.5.5— o canal 2 —canal 13 de la configuración triple del ADC—.

Para cambiar la frecuencia del reloj que disciplina el conjunto ADC —y en consecuencia el reloj de muestreo, ver sección 4.5.5— se utiliza el mensaje de tipo *MUESTREO*. En *carga1* de este tipo de mensaje, se almacena el dato del reloj, que es del tipo enumeración *RelojMuestreador* —ver sección 4.5.4—. Sus posibles valores son: *FREC36MHZ*, *FREC18MHZ*, *FREC12MHZ* y *FREC9MHZ* para los valores respectivos de 36 MHz, 18 MHz, 12 MHz y 9 MHz, que corresponden a las frecuencias de muestreo de 7,2 MHz, 3,6 MHz, 2,4 MHz, y 1,8MHz —ver sección 4.5.5—. La ejecución de la orden implícita en el mensaje se realiza a través del método de acceso *ponFrecMuestreador*, que admite como argumento el valor de *carga1* del mensaje y que modifica la variable miembro *muestreador*, que será la que tenga en cuenta el método *tareaFFT* al realizar el cambio de frecuencia.

Mediante el mensaje de tipo *TEMPORAL* se ordena a la aplicación el cálculo o no de la magnitud del espectro de los datos capturados. En *carga1* está el valor booleano que indica si se debe calcular la magnitud del espectro —valor falso— o no hacerlo, pasando al contexto gráfico los datos temporales capturados —valor verdadero—. La orden se realiza mediante la llamada al método de acceso *verTemporal,* que modifica el valor del atributo *temporal* acorde con su argumento —valor de *carga1*—. En la visualización temporal se produce el autoajuste —comprobando previamente que se ha podido poner a verdadero el atributo *temporal*— con la consecuente modificación de la escala y desplazamiento verticales.

Para modificar la posición vertical, el contexto gráfico manda a la aplicación el mensaje de tipo *POSICION_VERTICAL*, con el valor de la nueva posición en *carga1*. El método *interpretaMensaje* modifica el atributo *DesplazamientoVertical* —atributo que testeará *tareaFFT*— mediante el método de acceso *ponerPosicionVertical*. Este método no solo modifica el valor del atributo mencionado, sino que pone el atributo *escalaVerPosVerModificada* a verdadero para indicar a *tareaFFT* que debe especificar, en el siguiente mensaje que mande al contexto gráfico, que el modelo debe actualizar sus atributos *EscalaVertical* y *DesplazamientoVertical* acorde con los valores que se envíen en este segundo mensaje —ver sección 4.5.8—. El método *tareaFFT*, después de comprobar que el atributo *escalaVerPosVerModificada* está a verdadero y actuar en consecuencia, lo pone a falso, dejándolo en disposición de volver a ser modificado por un nuevo mensaje de este tipo o de los tipos *ESCALA_VERTICAL* o *POSICION_ESCALA_VERTICAL_TEMPORAL*.

El mensaje *ESCALA_VERTICAL*, instruye a la aplicación a modificar el factor por el que ha de multiplicar los datos tratados. Se hace a través de la llamada al método de acceso *ponerEscalaVertical*, que modifica el atributo *EscalaVertical* —el valor de la nueva escala viene en *carga1* del mensaje recibido—. Al igual que sucedía con el tipo de mensaje comentado con anterioridad, en este también se modifica el atributo *escalaVerPosVerModificada* con la misma finalidad.

El mensaje de tipo *POSICION_ESCALA_VERTICAL_TEMPORAL* es enviado por el contexto gráfico a la aplicación para restituir los valores previos de la escala y desplazamiento verticales, cuando anteriormente se ha mandado el mensaje *TEMPORAL* para visualizar los datos capturados por el conjunto *ADC*. El mantenimiento de los valores de escala y desplazamiento previos a la visualización temporal es responsabilidad del entrono gráfico, y con este tipo de mensaje se informa a la aplicación que ambos datos han sido mandados en este mensaje —en *carga1* va la escala y en *carga2* el desplazamiento—. En *carga3* está la información de si visualizar los datos temporales o no. Este tipo de mensaje también modifica el atributo *escalaVerPosVerModificada* con la misma finalidad que los dos anteriores.

Con el mensaje de tipo *AUTOPAN*, el entorno gráfico ordena a la aplicación si el autoajuste o seguimiento que se realicen con posterioridad, van a ser sobre los datos visualizados —*carga1* a valor verdadero— o sobre todo el registro —*carga1* con valor falso—. El atributo que dice a *tareaFFT* qué hacer al respecto, es *AutoPan*, y es cargado directamente con el valor de *carga1* en el método *interpretaMensaje*. Este mensaje no realiza una acción directa sobre la aplicación, sino que varía el comportamiento de los mensajes *AUTOAJUSTE* y *SEGUIMIENTO*.

Finalmente está el mensaje de tipo *SOLICITUD_DATOS*. Este tipo de mensaje es enviado por el entorno gráfico cuando no tiene otra orden que mandar a la aplicación. En cada *tick* en el modelo, se ha de mandar un mensaje hacia la aplicación. Este es el tipo de mensaje por defecto que posibilita la actualización de los datos de forma periódica. La información adicional que envía es el índice inicial —en *carga1*- y el índice final —en *carga2*— de los datos visualizados, para que si se ha activado el seguimiento, se pueda seguir autoajustando los datos aunque se varíe la posición horizontal.

4.5.10 Puesta en marcha de la tarea FFT.

A lo largo de todo este punto 4.5 se ha hablado de «aplicación» como sinónimo de instancia de la clase *DispositivoFFT*, pero no se ha indicado donde se declara el objeto de esta clase. Este objeto se decide crear en la del archivo *main.cpp*. Ello es debido a que *TouchGFX* crea el objeto de tipo HAL, que contiene la tarea gráfica, dentro de este archivo, y por tanto, es un lugar conveniente para declarar el objeto de tipo *DispositivoFFT* y encapsularlo dentro de una tarea del sistema operativo *FreeRTOS*. De esta forma, se tendrán localizadas, conjuntamente, tanto las definiciones como las declaraciones de las dos tareas de sistema que se ejecutarán: la tarea del entrono gráfico, y la tarea de la aplicación —método *tareaFFT* del objeto de la clase *DispositivoFFT*—. La parte del código del archivo main.cpp que hace referencia a la tarea de la aplicación, a las colas y semáforo, es el siguiente:

```
. . . //inclusión de cabeceras de RTOS y de inicialización de hardware

#include <gui/common/ComunicaTareas.hpp>
#include <gui/DispositivoFFT/DispositivoFFT.hpp>

xQueueHandle colaDeAplicacion_a_GUI;
xQueueHandle colaDeGUI_a_Aplicacion;
xSemaphoreHandle semaf_datos_disponibles;

DispositivoFFT fft;//objeto fft

#define configGUI_TASK_PRIORITY                 ( tskIDLE_PRIORITY + 3 )
#define configGUI_TASK_STK_SIZE                 ( 4000 )

. . . //tarea del entorno grafico para encapsular en tarea RTOS

static void FFTTask(void *parametros)
{
    fft.tareaFFT();
}

int main (void)
{
    . . . //rutinas de inicialización de hardware y de TOUCHGFX

    fft.configuraHardware(1);

    colaDeAplicacion_a_GUI = xQueueCreate( 4, sizeof(Mensaje));
    colaDeGUI_a_Aplicacion = xQueueCreate( 4, sizeof(Mensaje));

    semaf_datos_disponibles=xSemaphoreCreateBinary(); //se crea el semáforo
    xSemaphoreTake(semaf_datos_disponibles,0);// inicialmente no hay testigo

    . . . // tarea RTOS para el contexto gráfico

    xTaskCreate( FFTTask, (signed char*)"FFTTask",
                 configGUI_TASK_STK_SIZE,
                 NULL,
                 configGUI_TASK_PRIORITY,
                 NULL);


    vTaskStartScheduler();

    for (;;);
}
```

Código 4.142: Creación de la tarea de la aplicación, colas del sistema y semáforo DMA

Entre las líneas 6 a la 7 se declaran las dos colas del sistema —la que va en la dirección de la aplicación al entrono gráfico, línea 6, como la que tiene dirección del entorno gráfico a la aplicación, línea 7—. En la línea 8 se declara el semáforo que utilizará el *DMA* para indicar a la tarea de la aplicación —*tareaFFT*— que se ha completado la transferencia de datos de la captura —ver secciones 4.5.5 y 4.5.8—. En la línea 10 se declara el objeto o instancia de la clase *DispositifoFFT*, llamado *fft*. Este objeto no va a ser accedido desde fuera del archivo *main.cpp*, con lo que no es necesario declararlo de forma externa —con el especificador *extern*— en ningún archivo de cabecera. El método del objeto *fft* que actuará como tarea de la aplicación es *tareaFFT* y ha de encapsularse en una tarea del sistema operativo —líneas de la 17 a la 20— con un formato concreto —el argumento es un puntero de tipo *void* y no retorna ningún valor, ver capítulo 2—. Dentro del cuerpo de la función main, se realiza las inicializaciones pertinentes de todos estos elementos. Así, en la línea 26, se configura todo el hardware relacionado con la aplicación, especificando que el canal que se va a utilizar —entre los dos posibles— es el canal 1 —ver sección 4.5.5—. En las líneas 28 y 29 se crean las dos colas —para sentido—, cada una de ellas con cuatro elementos de tipo *Mensaje* —ver sección 4.5.7—. Estas colas se han de declara externamente ya que van a ser utilizadas desde varios lugares distintos —contexto gráfico y contexto de aplicación—. Ambas, son declaradas con el modificador *extern* en el archivo de cabecera ComunicaTareas.hpp, y este, incluido en el archivo de cabecera del contexto gráfico —Model.hpp—, y en el de la aplicación —DispositivoFFT.hpp—. En la línea 31 se crea el semáforo *semaf_datos_disponibles* —y declarado como *extern* en el archivo ComunicaTareas.hpp para ser utilizado dentro del objeto de tipo *DispositivoFFT*—. En la línea 32 se hace que el semáforo no tenga inicialmente testigo para que la aplicación espere a la finalización de la primera transferencia para intercambiar registros y realizar todo su proceso —ver sección 4.5.8—. La tares de RTOS, referente a la aplicación —objeto de tipo *DispositivoFFT*— es creada entre la líneas 36 a 40 —ver capítulo 2— después de haber creado la tarea del contexto gráfico —no mostrado en la porción de código—. Finalmente, el planificador, y por tanto, el sistema operativo, son puestos en marcha con la sentencia de la línea 43 y la función *main* entra en el necesario bucle infinito de la línea 45.

4.6 Creación del patrón MVC.

El patrón *MVC* —*MVP*— finalmente implantado, difiere de las secuencias mostradas en el capítulo 1. El esquema de secuencia utilizado en el proyecto es el siguiente:

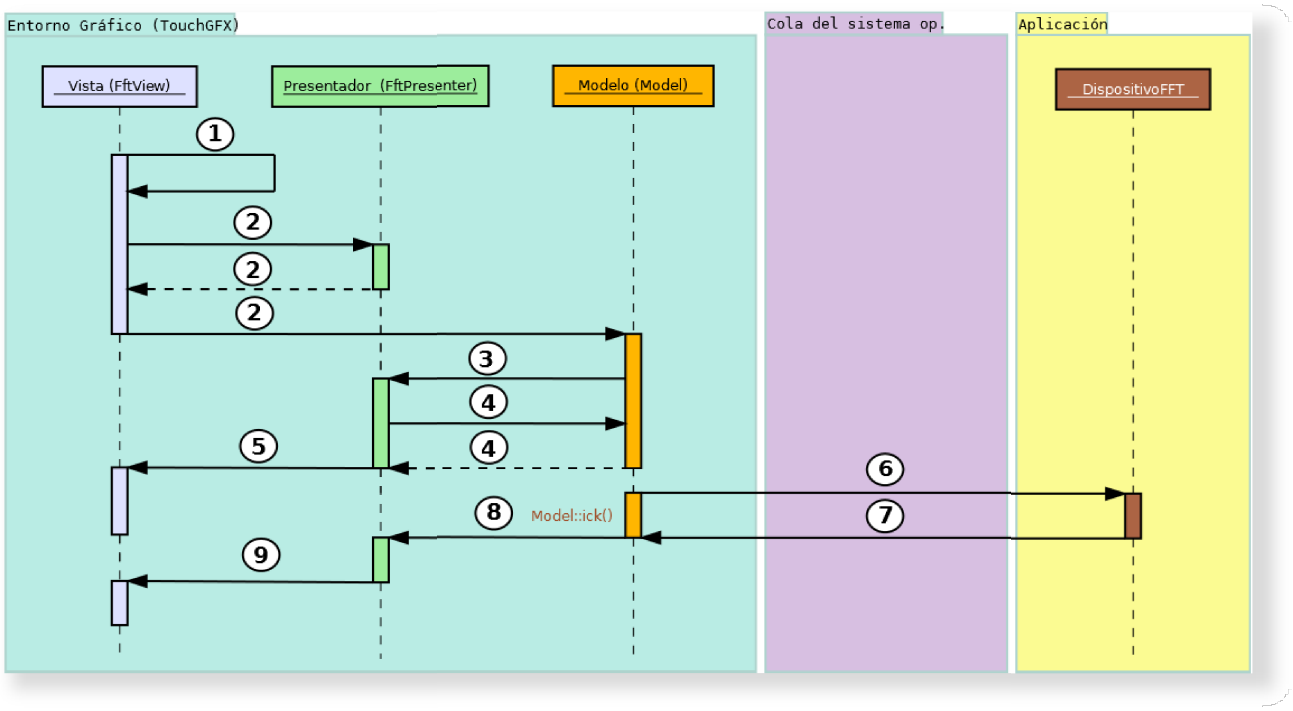


Ilustración 4.63: Patrón MVC implementado

Hay mensajes que son puramente de comunicación interna dentro del patrón *MVC* —los puntos marcados del 1 al 5—, mientras que otros —6 y 7 —también intervienen en la comunicación entre contextos —contexto del entorno gráfico (*TouchGFX*), y contexto de la aplicación (*DispositivoFFT*)—. Cuando un control gráfico en *TouchGFX* es manipulado —como pueda ser la pulsación sobre un botón—, se produce un evento disparado desde la capa *HAL*, identificando el componente responsable del disparo y el carácter del evento. Ya que el control que inicia el evento y el manipulador[31] que da respuesta al mismo, están localizados en el mismo lugar —la vista—, el mensaje puede ser tomado como recursivo —el mismo elemento que produce el evento, lo recibe; en este caso, la vista—. El manipulador puede cambiar varias propiedades del propio control o de otros controles relacionados en la vista, directamente —aunque, generalmente, serán solo las del propio control—, sin necesidad de intervención de otra entidad, como pueda ser el presentador o el modelo. Esto constituye el primer paso de la secuencia, mostrado con el número 1; el disparo del evento y el primer tratamiento del mismo. Sin embargo, estos cambios han de actualizarse en las propiedades del modelo que son reflejo de ciertas propiedades de los controles de la vista. Para ello, la vista solicita al presentador la dirección del modelo, ya que el presentador posee un puntero hacia el modelo, y una vez obtenido, la vista accede directamente a este —mensajes marcados con el número 2—. Al acceder al modelo, actualiza las propiedades pertinentes de este a través de los métodos de acceso a las mismas. Estos métodos del modelo, además de actualizar sus propiedades, actúan como un interfaz para el presentador, mandándole mensajes —marcados en el esquema como 3— que le indican únicamente qué propiedad del modelo acaba de actualizar la vista. Para conocer los valores de las propiedades del modelo al que han sido actualizados, el presentador ha de solicitar al modelo dicho valor a través de los métodos de acceso a los atributos de este último —mensajes marcados con el número 4—. Con estos valores, el

[31] Función o método de clase encargado de dar respuesta al evento producido, de forma similar a como lo hace una rutina de atención a una interrupción.

presentador puede actualizar otros controles gráficos de la vista que están relacionados con el cambio inicial del control que generó el evento. Así, por ejemplo, dos controles, en lugares diferentes —en menús diferentes—, que ambos modifican una misma propiedad en la aplicación de usuario, pueden ser actualizados a la vez cuando cualquiera de ellos es cambiado. Esta es la principal utilidad del presentador: centralizar los cambios en la vista a consecuencia de la interacción con el usuario. La vista está dedicada a contener los elementos gráficos mientras que el modelo mantiene el estado de ciertas propiedades de los controles de la vista para poder restituirlas en caso de cambios entre pantallas —cambios entre pares de vista-presentador—. Por otro lado, y de forma independiente, está la comunicación entre el entrono gráfico —*TouchGFX*— y la aplicación de usuario —*DispositivoFFT*— a través de las colas del sistema operativo —ya visto en la sección 4.5—. Para ello, en el modelo existe el método *tick*; método que es llamado cuando se actualiza la pantalla —idealmente en el sincronismo vertical—. En cada llamada a este método se manda un mensaje —marcado como 6 en el esquema—, a través de las colas del sistema, que indica a la aplicación de usuario que puede mandar los datos de la última adquisición y procesado de datos —mensaje marcado como 6 en el esquema—. Este mensaje, por defecto, significa «petición de datos», aunque, debido a la actividad interna de mansajes del entorno gráfico —mensajes del 1 al 5—, puede ser modificado para que actualice ciertos valores de las propiedades del objeto que constituye la aplicación de usuario —*DispositivoFFT*—, pero en cualquier caso, implica la petición de nuevos datos —ver sección 4.5—. La aplicación de usuario, ante una petición del entorno gráfico, genera un mensaje, con los nuevos datos procesados, a través de las colas del sistema —mensaje marcado como 7 en el esquema—, y actualiza sus propiedades. El mensaje con los nuevos datos llega al modelo —marcado como 8 en el esquema—, y este genera un mensaje hacia el presentador con ellos. El presentador lo retransmite hacia vista —mensaje marcado con 9—, donde los datos son finalmente mostrados en pantalla.

4.6.1 Modificación de la vista, el presentador y el modelo para la implementación real del patrón MCV utilizado.

El esquema presentado al inicio de esta sección —4.6—, no es realizable tal y como está implementado el patrón *MVC* en *TouchGFX*. Para que funcione, es necesario realizar dos modificaciones importantes. La primera de ellas consiste en dar acceso directo al modelo por parte de la vista. Hay que recordar que la única entidad que tiene acceso al modelo es el presentador a través de un puntero que posee como atributo. Esta primera modificación consiste en crear un método público en el presentador, llamado *obtenerModelo*, que devuelva este puntero Así, llamado desde la vista, se podrá acceder directamente al modelo —mensaje marcado como 2 en el esquema de secuencia al inicio de esta sección, 4.6—. El código de esta nueva función es la siguiente:

```
1  class FftView; //declaración adelantada de la vista
2
3  class FftPresenter : public Presenter, public ModelListener
4  {
5      public:
6          . . .
7          /**
8           *  Método que retorna el puntero del modelo
9           *  "model", declarado en la clase antecesora
10          *  "Presenter"
11          */
12         virtual Model* obtenerModelo(void)
13
14         {
15             return model;
16         }
17
18     private:
19         . . .
20 };
```

Código 4.143: Método de acceso directo al Modelo

La segunda modificación en el patrón hace referencia a la relación entre el presentador y la vista. El presentador tiene acceso a la vista desde una referencia interna llamada *view*. Sin embargo, solo tiene acceso a la parte pública de la vista. Se recuerda que todos los controles gráficos son declarados en la parte privada de la vista, con lo que, para que el presentador tenga acceso a sus propiedades, se han de crear métodos de acceso a las mismas en la parte pública de la vista. Esto puede resultar muy engorroso a la hora de escribir código o su posterior lectura e interpretación por parte del programador. Además, supone una duplicación de código, ya que cada control gráfico, de forma interna, ya posee métodos de acceso a sus propiedades. Para solventar este problema se hace que la clase del presentador sea amiga de la clase de la vista, de esta forma, el presentador tendrá acceso a las partes privadas de la vista:

```
1  class FftView : public View<FftPresenter>
2  {
3
4      public:
5          . . .
6
7      private:
8
9          friend class FftPresenter;//permite al presentador el acceso
10                                    //a las partes privadas de la vista
11         . . .
12 };
```

Código 4.144: Acceso a la parte privada de la vista por el presentador

Para que el modelo pueda mandar mensajes al presentador, se crea en el archivo de cabecera ModelListener.hpp —que contiene la clase *ModelListener*— una función virtual con cuerpo vacio que el presentador ha de redefinir:

```
1  class ModelListener
2  {
3      public:
4          . . .
5          virtual void interpretePresentador(ComandosPresentador comando) {}
6          . . .
7
8      protected:
9          Model* model;
10 };
```

Código 4.145: mecanismo de mensajes del modelo al presentador

De esta forma se asegura que, ante un posible cambio del par vista-presentador, al ser llamado a este método, siempre disponga de un cuerpo, aunque sea vacío.

4.6.2 El Modelo: atributos y métodos de acceso

Como ya fue comentado en el capítulo 1, el modelo es la entidad que sirve para mantener el estado de las vistas entre transiciones. Esto significa que ha de guardar la información sensible de cada vista que deba ser restituida al volver a ella después de un cambio a otras vistas. Qué es lo que se debe guardar y qué no, entre transiciones, es una decisión del programador. Estos datos guardados están representados por los atributos de la clase *Model* —datos miembro de la clase—. Pero el modelo también sirve para la comunicación con la aplicación final de usuario —la aplicación realmente útil, que no pertenece al entorno gráfico— a través del método *tick*. Por ello, el modelo poseerá tanto atributos que son referentes a las vistas utilizadas, como atributos que lo son a la aplicación final. Dentro de los atributos referentes a la vista, se pueden distinguir aquellos que tienen repercusión en la aplicación final y aquellos que no —son puramente modificaciones estéticas—. Los atributos utilizados en el modelo del proyecto, son los siguientes:

```
1  const float CGs[]={1.0f, 0.5f, 0.54f, 0.36f, 0.5f};
2  . . .
3  class Model
4  {
5      public:
6          Model() : modelListener(0)
7          { . . . }
8                      . . .
9      protected:
10
11         ModelListener* modelListener;
12
13         bool adquiere;
14
15         Mensaje mensaje_saliente;
16         Mensaje mensaje_entrante;
17
18         DispositivoFFT::TiposVentana ventana;
19         DispositivoFFT::RelojMuestreador muestreador;
20         Gamas gamas;
21         uint16_t IndiceInicial;
22         uint16_t IndiceFinal;
23         bool AutoPan;
24         bool Seguimiento;
25         bool EscalaLogaritmica;
26         uint8_t Canal;
27         bool Temporal;
28         float DeltaFrecMuestreo;
29
30         int16_t DespHor;
31         float EscalaHorizontal;
32
33         float DesplazamientoVertical;
34         float EscalaVertical;
35
36         float nivelPorPixelFFT;
37         float nivelPorDivision;
38         float nivelMenorEnPantalla;
39         float nivelMayorEnPantalla;
40         float nivelEjeVertical;
41         float CG;
42
43         float frecuenciaPorPixel;
44         float frecuenciaPorDivision;
45         float frecuenciaMenorEnPantalla;
46         float frecuenciaCentral;
47         float frecuenciaMayorEnPantalla;
48         float frecuenciaSpan;
49
50     private:
51
52         void determinaConversionesNivel(void);
53 };
```

Código 4.146: Atributos del Modelo del proyecto

El atributo *modelListener* —línea 11—, es el puntero al presentador actual —*ModelListener* es una clase interface que implementa el presentador—. A través de él, el modelo manda mensajes al presentador para que este último actualice la vista en consecuencia.

Para poder controlar la adquisición, en la aplicación final de usuario —clase *DispositivoFFT*— se provee el atributo *adquiere* —línea 13—. Aunque la modificación de este atributo tiene repercusión en la aplicación de usuario, el modelo realmente no manda ningún mensaje específico. De hecho, el poner este atributo a valor falso, provoca que el modelo no mande ningún tipo de mensaje, lo que produce que no haya sincronización con la aplicación final —esta queda bloqueada por un semáforo—. Aunque este atributo no induce comunicación con la aplicación final, debido al comportamiento que ocasiona en ella, se le puede catalogar de atributo modificado por la vista que tiene repercusión en la aplicación de usuario. Este atributo es cambiado por el botón de opción *botonActivacionAdquisicion* de la vista —que aparece con el nombre «Adquisición» en el menú Adquisición— y por el botón marcador *ledActivador*, que aparece a modo de led verde, sobre la pantalla, parpadeando cuando la adquisición está detenida.

La comunicación entre contextos —el modelo, perteneciente al entorno gráfico y la aplicación final—, se realiza mediante los atributos *mensaje_saliente* y *mensaje_entrante* —líneas 15 y 16—. Estos son de uso exclusivamente del modelo —sin acceso para la vista— y su estructura y modo de utilización fueron comentados en la sección 4.5.7.

El grupo de atributos entre las líneas 18 y 27, son reflejo de atributos en la aplicación final, y son modificados desde la vista. De hecho, alguno de los tipos utilizados están definidos en la clase de la aplicación —líneas 18 y 19—. El atributo *ventana* —línea 18— especifica el tipo de ventana a utilizar en la FFT en la aplicación final —las posibles ventanas son: *rectangular*, *hanning*, *hamming*, *blackman-harris* y *bartlett*, ver sección 4.5.3 y 4.5.6—. Este atributo es modificado por los botones de opción *botonesVentanas[]* del menú «Ventanas» y por el botón de estado *BotonesEstado[0]* del menú «Rápido». Para poder modificar la velocidad de captura de datos, por parte del ADC en la aplicación final, se provee el atributo *muestreador* —línea 19—. Se puede elegir entre cuatro posibles relojes —36 MHz, 18 MHz, 12 MHz y 9 MHz, ver sección 4.5.4 y 4.5.5— y es modificado por el botón de estado *BotonesEstado[6]* del menú «Rápido» así como por el elemento de selector *elementosFs[]* del menú «Adquisición». El cambio de gama de color de relleno entre la gráfica y la ordenada, se realiza a través del cambio de valor del atributo de tipo enumeración *gamas* —línea 20— entre los posibles que son: JET, COOL, HOT, OCEAN, RAINBOW, WINTER y AUTUMN. Es modificado por los botones de opción *botonesGamas[]* del menú «Esquemas» y por el botón de estado *BotonesEstado[5]* del menú «Rápido». Para poder autoajustar la amplitud, solo a los datos visualizados en pantalla, se utilizan los dos siguientes atributos *IndiceInicial* e *IndiceFinal* —líneas 21 y 22—, que especifican los índices límites visualizados en pantalla de la serie —que contiene la FFT—. Estos dos atributos son especificados de forma indirecta mediante la acción conjunta de los controles gráficos *botonActivacionAutoPan* de tipo *RadioButton*, en el menú «Adquisición», y el botón *botAutoajuste* del menú «Ventanas» o el botón de estado *BotonesEstado[2]* del menú «Rápido». El atributo *AutoPan* —línea 23—, especifica si el autoajuste de nivel se hace teniendo en cuenta los dos atributos anteriores —los índices límite visualizados— o todo el registro. Este atributo es modificado por el *botonActivacionAutoPan* de tipo *RadioButton* del menú «Rápido». El atributo *Seguimiento* —línea 24— especifica si el auto ajuste es realizado de forma continua en cada actualización de pantalla. Se modifica mediante el control *BotonesEstado[3]*, de tipo botón de estado del menú «Rápido». Para controlar si la visualización de los datos se realiza de forma lineal o logarítmica, se utiliza el atributo *EscalaLogaritmica* —línea 25—. Es modificado por el botón *RadioButton botonActivacionLogaritmo* del menú «Adquisición» o por el botón de estado *BotonesEstado[1]* del menú «Rápido». El atributo *Canal* —línea 26—especifica el canal de origen a realizar la FFT —de entre los dos posibles— y es modificado por los *RadioButtons botonCanal1* y *botonCanal2* del menú «Adquisición» o por el botón de estado *BotonesEstado[4]* del menú «Rápido». Para poder visualizar los datos temporales tal y como son adquiridos —sin realizar la FFT— está el atributo *Temporal* —línea 27—. Es modificado por *botonActivacionTemporal* de tipo *RadioButton*.

El atributo *DeltaFrecMuestreo* es utilizado internamente por el entrono gráfico para poder determinar la frecuencia de cada dato del array, que contiene la FFT, a partir del índice de cada uno. Es actualizado de forma indirecta al cambiar atributo *muestreador*. Su cálculo se realiza mediante la siguiente expresión:

$$\Delta F = \frac{PCLK2 / preescalador}{n\_datos \cdot ciclos\_entre\_muestras} \tag{4.59}$$

Donde *PCLK2* es el reloj para el bus *APB2* del núcleo *ARM* —ver sección 4.2—, *preescalador*, es el divisor del reloj —de los cuatro posibles, que son 2, 4, 6 y 8— que disciplina el conjunto de los tres ADCs trabajando de forma simultánea —ver sección 4.5.5—, *n_datos*, es el número de datos utilizados en el cálculo de la FFT, en este caso concreto es de 2048, y *ciclos_entre_muestras*, los ciclos necesarios para que el conjunto ADC suministre una nueva muestra —que para este caso es de cinco ciclos, ver sección 4.5.5—.

Los dos siguientes atributos, *DespHor* y *EscalaHorizontal* —líneas 30 y 31—, solo afectan a la vista sin repercusión en la aplicación final de usuario. El cambio de *DespHor* y *EscalaHorizontal* provoca el desplazamiento horizontal de la gráfica y el número de datos a visualizar, respectivamente, sin producir alteración en el comportamiento de la aplicación final. La aplicación final calcula la FFT de todos los datos, y es a través de estas propiedades donde se determina qué parte de estos datos mostrar. *DespHor* es modificado por botones de retención —*HoldableButton*— *decFrecCentral* e *incFrecCentral*, pertenecientes al primer contenedor deslizable del menú «Controles» y por el evento de arrastre —*DragEvent*— del control *areaTocable* de tipo *TouchAreaMio* —ver sección 4.4.1.2—. *EscalaHorizontal* es modificado por los botones *decSpan* e *incSpan* del segundo contenedor deslizable del menú «Controles». La modificación de ambos atributos va a producir un mensaje hacia el presentador que provocará que se vuelvan a calcular todos los datos referentes a frecuencia como son la frecuencia central —la que aparece en mitad de pantalla— o el *span* —ancho de frecuencia visualizado horizontalmente en pantalla—, para ello se hará uso del valor del atributo *DeltaFrecMuestreo* antes comentado.

Los atributos *DesplazamientoVertical* y *EscalaVertical* son actualizados, principalmente, por la aplicación final de usuario —el objeto de la clase *DispositivoFFT*— cuando se realiza un «autoajuste» o «seguimiento». Se trata de atributos reflejo de los existentes en la aplicación final. Su finalidad es la de ajustar los datos a los límites de 0 a 255 para que entren dentro de datos de tipo uint8_t. La actualización de estos valores, desde la aplicación final, se realiza en el método *tick* del modelo. La expresión de la corrección que estos atributos hacen sobre los datos —en el objeto de la clase *DispositivoFFT*—, es la siguiente:

$$\begin{aligned} \text{Datos}_{\text{corregidos}} &= \frac{255}{MAX - MIN} \cdot Datos - \frac{255}{MAX - MIN} \cdot MIN \\ \text{Datos}_{\text{corregidos}} &= EscalVer \cdot Datos - DesVer \end{aligned} \tag{4.60}$$

Donde *MAX* es el mayor de los datos del registro —o de los datos visualizados, dependiendo si se usa *Autopan* o no—, y *MIN*, es el menor de los datos del grupo de datos contemplado. Estos dos atributos pueden ser modificados por el usuario. *DesplazamientoVertical* se modifica a través de los controles *decDesplazamiento* e *incDesplazamiento* —de tipo *HoldableButton*—, pertenecientes al tercer contenedor deslizable del menú «Controles». *EscalaVertical* se modifica a través de los botones *decAmplitud* e *incAmplitud* del cuarto contenedor deslizable del menú «Controles», o mediante los botones de ocultación —*TBotonOcultado*— *botonOcultado_masV* y *botonOcultado_menosV*.

El siguiente grupo de atributos —líneas de la 36 a la 41— hace referencia a conversiones de nivel. Así, *nivelPorPixelFFT* establece el factor multiplicativo para pasar de pixeles a unidades de nivel —lineal o logarítmico—, *nivelPorDivision* hace lo propio para establecer el nivel de una división vertical de la rejilla —retícula—, *nivelMenorEnPantalla*, establece el nivel correspondiente a la parte inferior de la pantalla, mientras que *nivelMayorEnPantalla* lo hace para la parte superior. La diferencia entre estos dos últimos atributos es el valor del atributo *nivelEjeVertical*. Para poder corregir la lectura de nivel, cuando hay una ventana temporal aplicada —ver sección 4.5.3— se utiliza el atributo *CG*. Este contendrá uno de los valores del *array CGs* —línea 1—, que corresponderá al valor de la ganancia de coherencia a aplicar. Las ganancias que aparecen en este *array* son, respectivamente, para las ventanas *rectangular*, *hanning*, *hamming*, *blackman*-harris y *bartlett*. Estos atributos, a excepción de *CG*, son actualizados mediante la llamada al método privado *determinaConversionesNivel* —línea 52—, por parte del método *tick* cuando se produce una modificación de la escala o desplazamiento vertical —a consecuencia de un autoajuste o que el usuario modifique estos parámetros—. El método *determinaConversionesNivel* será explicado en la sección 4.6.4. *CG* es actualizado cuando se modifica el atributo ventana —línea 18—.

Finalmente, se encuentran los atributos relacionados con la conversión de frecuencia —líneas de la 43 a la 48—. El atributo *frecuenciaPorPixel* establece el valor del incremento de frecuencia de punto a punto de pantalla, *frecuenciaPorDivision* determina cuánto mide el ancho de cada cuadro de la retícula, *frecuenciaMenorEnPantalla* establece el valor de frecuencia en la zona izquierda de las pantalla mientras que *frecuenciaMayorEnPantalla* lo hace en la parte derecha. La diferencia entre ambos valores es el contenido del atributo *frecuenciaSpan*. La frecuencia en la mitad de pantalla la determina *frecuenciaCentral*. Todos estos atributos son actualizados por el presentador —*FftPresenter*— cuando se produce una modificación del atributo *DespHor*. Cuando es actualizado el atributo *EscalaHorizontal,* todos los atributos referentes a conversión de frecuencia son actualizados a excepción de *frecuenciaCentral*.

De forma general, a cada atributo le corresponde un par de métodos públicos de acceso. Uno de ellos será para la lectura, cuyo nombre estará constituido por el prefijo «obtener», seguido, generalmente, del nombre del atributo al que se quiere acceder —sin dejar espacios—. El otro será para la escritura, cuyo nombre estará compuesto del prefijo «poner» seguido, generalmente, del nombre de la propiedad a modificar —igualmente sin dejar espacios—. De forma concreta, no todos los atributos tendrán dos métodos de acceso, los atributos que se desee de solo lectura, solo tendrán el método «obtener», mientras que los de solo escritura, el método «poner». Así los atributos *AutoAjuste*, *IndiceInicial* e *IndiceFinal*, serán de solo escritura, mientras que *DeltaFrecMuestreo*, *NivelPorPixelFFT*, *NivelPorDivision*, *NivelMenorEnPantalla*, *NivelMayorEnPantalla* y *NivelEjeVertical* serán de solo lectura. El resto son de lectura-escritura.

Los métodos de lectura se limitarán a devolver el valor del atributo al que hacen referencia. En cambio, los métodos de escritura tienen una estructura más compleja. Genéricamente, los métodos de escritura siguen tres pasos, que son:

1) Actualizar la propiedad en el modelo y otras propiedades dependientes de esta.
2) Mandar al presentador que actualice en la vista todos los widget a los que afecte el cambio de propiedad.
3) Crear el mensaje saliente para la tarea de la aplicación final y que esta actúe en consecuencia al cambio del valor de la propiedad.

Todos los métodos de escritura no tendrán que cumplir estos tres puntos. Por ejemplo, los atributos con solo repercusión en la vista —como *DespHor* y *EscalaHorizontal*— no cumplirán el tercer paso. Otros atributos no necesitan cumplir el segundo ya que la orden de actualizar la vista, a través del presentador, no es lanzada desde el método de escritura. Este es el caso de las propiedades *DesplazamientoVertical* y *EscalaVertical*, donde la orden de actualizar la vista —debido al cambio de estos atributos—, es realizada en el método *tick* si el mensaje proveniente de la aplicación final —*DispositivoFFT*— le indica que se han producido cambios en la escala o desplazamiento vertical en la aplicación. Existen métodos de escritura no tienen que cumplir el primer paso ya que no hacen referencia a ningún atributo del modelo. Este es el caso del método *ponerAutoAjuste*, que se limita a actualizar la vista —paso 2— y a modificar el mensaje de salida —paso 3—. Pero la mayoría de atributos cumplen estos tres pasos. Como ejemplo, se muestra el código del método de escritura de la propiedad *muestreador*:

```
1  void Model::ponerMuestreo(DispositivoFFT::RelojMuestreador muestreador)
2  {
3    // 1) Actualizar la propiedad en el modelo y otras propiedades dependientes de esta
4      this->muestreador=muestreador;
5      uint8_t preescalador;
6      uint8_t ciclos_entre_muestras=5;//mirar archivo excel : simulation.xls
7      switch (muestreador)
8      {
9          case DispositivoFFT::FREC36MHZ:
10             preescalador=2;
11             //DeltaFrecMuestreo=36;
12             break;
13         case DispositivoFFT::FREC18MHZ:
14             //DeltaFrecMuestreo=18;
15             preescalador=4;
16             break;
17         case DispositivoFFT::FREC12MHZ:
18             //DeltaFrecMuestreo=12;
19             preescalador=6;
20             break;
21         case DispositivoFFT::FREC9MHZ:
22             //DeltaFrecMuestreo=9;
23             preescalador=8;
24             break;
25     }
26     DeltaFrecMuestreo=(float)(((((float)SystemCoreClock/1e3f)/2)/preescalador)/\
27                       ciclos_entre_muestras)/(float)2048
28
29   // 2) Mandar al presentador que actualice en la vista todos los widget a los que
30   //    afecte el cambio de propiedad
31     modelListener->interpretePresentador(ACTUALIZA_PROP_FREC_MUESTREO);
32   // 3) Crear el mensaje saliente para la tarea de la aplicación y que esta
33   //    actúe en consecuencia al cambio del valor de la propiedad
34     mensaje_saliente.carga1.ValorInt=(DispositivoFFT::RelojMuestreador)this->muestreador;
35     mensaje_saliente.evento&=~MASCARA_TIPO_EVENTO;
36     mensaje_saliente.evento|=MUESTREO;
37 }
```

Código 4.147: Ejemplo de método de escritura del modelo

Entre la líneas 4 y 27 se realiza el primer paso. En la línea 1, se actualiza el propio atributo, mientras que en la 26 y 27 se actualiza la propiedad *DeltaFrecMuestreo* debido al cambio del atributo *muestreador.* La propiedad *DeltaFrecMuestreo* es actualizada siguiendo la expresión (4.56) mostrada más atrás. En la línea 31 se manda al presentador el mensaje *ACTUALIZA_PROP_FREC_MUESTREO* para que este actúe en consecuencia actualizando los controles oportunos de la vista —este mecanismo será mostrado en la sección 4.6.5—. Esto representa el paso 2. Finalmente, en este método, se realiza el tercer paso consistente en la modificación del mensaje saliente, especificando que se trata de un dato de muestreo —línea 36—, cuyo valor va en la carga1 del mensaje —ver sección 4.5.7 para la especificación de la estructura de los mensajes entre entorno gráfico y aplicación—. El diagrama de secuencia en el que está involucrado la llamada a este método, es el siguiente:

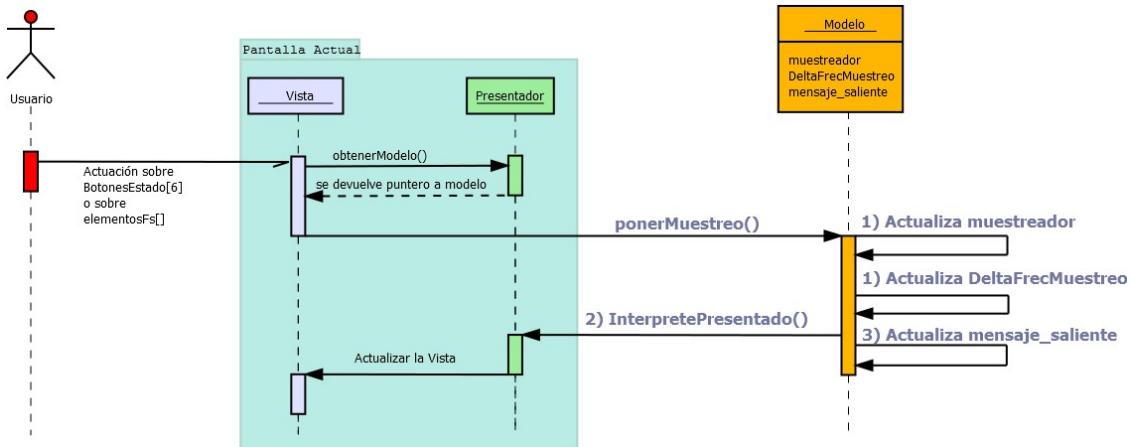


Ilustración 4.64: Diagrama de secuencia de método de escritura del modelo

Los mensajes de secuencia que han sido destacados en este esquema son los referentes al método de escritura del modelo —mostrados con letra más grande y en negrita—. De todas formas, se ha mostrado la secuencia completa que provoca la llamada al método de escritura del modelo *ponerMuestreo* —mostrada en el código anterior—, especificando, sobre el esquema, los tres pasos que ha de cumplir. Toda la secuencia se inicia cuando el usuario acciona el botón de estado *BotonesEstado[7]*, del menú «Rápido», o elige un elemento de selector *elementosFs[]* determinado, del menú «Adquisición». Esto provoca que la vista solicite al presentador el puntero hacia el modelo. Una vez obtenido, la vista ordena al modelo a actualizar su atributo interno *muestreador*, llamando a su método público *ponerMuestreador*() —método de escritura del modelo—. La ejecución de este método induce la realización de los tres pasos —mostrados en el esquema—. En el paso 1 se actualiza el propio atributo *muestreador* con el valor del argumento del método *ponerMuestreador*() que la vista ha puesto. En este mismo paso se actualiza otro atributo relacionado con el cambio del atributo anterior: *DeltaFrecMuestreo*. Una vez realizado este paso, se lleva a cabo el segundo: la llamada del método del presentador *InterpretePresentador().* Este método del presentador recibe como argumento un tipo enumerado que le indica qué actualizar en la vista. Para realizar esta actualización, necesita preguntarle al modelo sobre el valor de los atributos actualizados —todo esto no está mostrado en el diagrama de secuencia ya que será explicado en la sección 4.6.5—. El modelo realiza su tercer paso modificando el mensaje de salida que será enviado a la aplicación final para que esta actualice su atributo *muestreador* y realice las operaciones oportunas. El mensaje de salida será mandado por el método *tick*, un método del modelo que se ejecuta automáticamente en cada actualización de pantalla. Se recuerda que, aunque no se tenga nada que actualizar en la aplicación final, el modelo, a través del método *tick*, enviará el mensaje por defecto a la aplicación para que esta devuelva los nuevos datos procesados. El envío de un mensaje distinto al por defecto, además de instruir a la aplicación para actualizar alguna propiedad, implícitamente también significa petición de datos nuevos.

A parte de los métodos de acceso a los atributos y el método privado *determinaConversionesNivel* —que será visto en la sección 4.6.4—, el modelo dispones del método *tick* —previamente comentado y descrito en la sección 4.6.3—, el método *actualizaTodaLaVista*, cuya misión es la de cargar con valores por defecto los controles de la vista y será descrito en la sección 4.6.4, y el método *ponerEscalaDesplazamientoVerticalTemporal*, que sirve para modificar a la vez las propiedades de escala y desplazamiento vertical, así como especificar si se desea visualizar los datos en el dominio del tiempo. Esta última función es utilizada al restituir la visualización en el dominio de la frecuencia, ya que se necesitan realizar estos tres cambios de atributos tanto en el modelo como en la aplicación final.

4.6.3 Método «tick» del modelo: comunicación con la aplicación final y el presentador —ModelListener—

Al exponer el desarrollo de la aplicación —sección 4.5—, se mostró el siguiente esquema, resaltando aquellas partes del mismo relacionadas con ella. Es este caso, se vuelve a mostrar, destacando las partes involucradas con el entorno gráfico:

Ilustración 4.65: Detalle de la comunicación del entorno gráfico con la aplicación

Como parte oscurecida aparece la aplicación —tratada en el punto 4.5—. El elemento más importante de este esquema, tratado en esta sección, es el método *tick* del modelo. Este gestiona la comunicación entre el entorno gráfico y la aplicación —flecha bidireccional vertical azul—, a través de las colas del sistema operativo —vistas en la sección 4.5.7—, y la comunicación entre el modelo y el resto del entrono gráfico —flecha bidireccional verde entre el modelo y el presentador—. El modelo se comunica con el presentador a través de un puntero de tipo *ModelListener*, un interface[32] que el presentador implementa. El presentador se comunica con el modelo a través de un puntero al mismo, que el interface *ModelListener* posee. El presentador se comunica con la vista a través de una referencia que este posee como atributo. Este es el camino de comunicación del modelo con la vista: primero se comunica con el presentador, y este con la vista, no hay comunicación directa entre el modelo y la vista. La vista, por su parte, se comunica con el presentador a través de un puntero, pero no tiene acceso directo al modelo. Esta es la comunicación establecida en TouchGFX para el entorno gráfico —para más información ver la sección 1.9—, sin embargo, para facilidad de acceso entre estas entidades, se han realizado una serie de modificaciones que fueron descritas en la sección 4.6.1, siendo la más importante de ellas, la creación de un método en el presentador que facilita a la vista el puntero al modelo, y de esta forma la vista tenga acceso directo al modelo —ver sección mencionada para más información—. Pero como ha sido mencionado, el punto más importante de la comunicación es el método *tick*, ya que es el centro de emisión y recepción de mensajes. Por una parte, genera mensajes hacia la aplicación de forma periódica —en cada actualización de pantalla—. Estos mensajes solicitan a la aplicación nuevos datos y en ellos —los mensajes—, está implicado el resto del entrono gráfico —la vista, concretamente—. Por otra, recibe los mensajes de la aplicación, con nuevos datos, y los redirige al presentador para que este los mande a la vista.

[32] Derivación de una clase desde otra —el interface— para que sea vista desde el exterior como perteneciente a la clase del interface. Esto posibilita que se acceda a la clase original a través de un puntero del tipo del interface —polimorfismo—

El código suministrado al método «tick» del modelo, en este proyecto, es el siguiente:

```
1  void Model::tick()
2  {
3    if(adquiere)
4      if(modelListener)
5       {
6        if(mensaje_saliente.evento==SOLICITUD_DATOS)
7         {
8          mensaje_saliente.carga1.ValorInt=IndiceInicial;
9          mensaje_saliente.carga2.ValorInt=IndiceFinal;
10        }
11
12       xQueueSend(colaDeGUI_a_Aplicacion, (void *)&mensaje_saliente, (portTickType)10);
13
14       ADC_ClearFlag(ADC1, ADC_FLAG_OVR);
15       ADC_ClearFlag(ADC2, ADC_FLAG_OVR);
16       ADC_ClearFlag(ADC3, ADC_FLAG_OVR);
17
18       if(xQueueReceive( colaDeAplicacion_a_GUI, &mensaje_entrante, (portTickType)15))
19        {
20         switch((mensaje_entrante.evento & MASCARA_TIPO_EVENTO))
21          {
22           case DATOS_ESPECTRO:
23              modelListener->actualizaSerieFFT(
24                           (unsigned char*)mensaje_entrante.carga1.ValorInt,
25                           mensaje_entrante.carga2.ValorFloat,
26                           mensaje_entrante.carga3.ValorFloat,
27                           mensaje_entrante.carga4.ValorInt);
28
29              if((mensaje_entrante.evento & MASCARA_TIPO_SUBEVENTO) ==\
30                                                    ESCALA_VER_DES_VER_MODIFICADOS)
31               {
32                DesplazamientoVertical=mensaje_entrante.carga2.ValorFloat;
33                EscalaVertical=mensaje_entrante.carga3.ValorFloat;
34                determinaConversionesNivel();
35                modelListener->interpretePresentador(ACTUALIZA_PROP_ESCALA_VERT);
36                modelListener->interpretePresentador(ACTUALIZA_PROP_DESPLA_VERT);
37               }
38           break;
39          }
40         if((mensaje_entrante.evento & MASCARA_DATO_EVENTO) == DATO_RECIBIDO)
41          {
42            mensaje_saliente.evento=SOLICITUD_DATOS;
43          }
44        }
45      }
46 }
```

Código 4.148: Método «tick»

Toda la ejecución del método está condicionada por el valor —que sea verdadero, línea 3— del atributo *adquiere* —ver sección anterior 4.6.2—. Igualmente, está condicionada por la existencia del puntero al presentador *modelListener* —línea 4—. Experimentalmente se comprueba la presencia de problemas, en la ejecución de todo el sistema, si no existe esta condición. En cada ejecución de este método —ejecutado de forma automática en cada actualización de pantalla—, lo primero que se realiza es crear un mensaje saliente por defecto que solicite a la aplicación final nuevos datos —líneas de la 6 a la 10—. Este, va a tener como tipo de mensaje *SOLICITUD_DATOS*, para que el intérprete de la aplicación instruya a la tarea de la aplicación —tareaFFT— que mande nuevos datos —ver sección 4.5.9—. Como datos de este mensaje, se rellenan *carga1* y *carga2* con los valores respectivos del índice inicial y final de los datos visualizados. Es posible que la vista modifique este mensaje de salida para que no sea el por defecto. Ello ocurre cuando la vista modifica algún atributo del modelo que tenga repercusión en la aplicación final —ver sección 4.6.2—. En cualquier caso, el mansaje saliente siempre será entendido por la aplicación como solicitud de nuevos datos, además del significado propio del mensaje que se deriva de su tipo. Si el mensaje saliente ha sido modificado, el método *tick* ha de reiniciarlo para que en su próxima llamada sea el por defecto —a no ser que la vista vuelva a modificar un atributo del modelo con repercusión en la aplicación—. Para ello, antes de reiniciar el mensaje saliente, el método *tick* ha de tener constancia que el último mensaje fue recibido en la aplicación. Ello se consigue en la parte del código entre las líneas 40 a 43. Una vez recibido el mensaje por parte de la aplicación, aparte de ejecutar el código correspondiente al significado del mensaje, devuelve al modelo el mensaje de tipo

DATOS_ESPECTRO, con los datos de la FFT de los nuevos datos capturados y el dato de evento *DATO_RECIBIDO*, que indica al modelo que puede reiniciar el nuevo mensaje a *SOLICITUD_DATOS* —mensaje por defecto—. Una vez determinado el mensaje a mandar a la aplicación —ya sea el por defecto o el establecido por la vista—, se procede a su envío por la cola del sistema operativo *colaDeGUI_a_Aplicacion* —línea 12—. Para ello se utiliza la función del sistema operativo *xQueueSend*, donde se especifica la cola a utilizar, el mensaje a enviar —*mensaje_saliente*— y el tiempo que se está dispuesto a esperar el envío antes de abortar. Este tiempo se especifica en «Ticks» del sistema operativo, en este caso es de 10. El tipo de este dato ha de ser *portTickType* que no es más que un sinónimo para tipo *long* — 32 bits—. Ya que la constante *configTICK_RATE_HZ* está definida al valor de 1000 en el archivo FreeRTOSConfig.h, cada *Tick* del sistema operativo se produce en cada milisegundo, y por tanto, el dato especificado en el tercer argumento de la función *xQueueSend*, ya está expresado en milisegundos. Otra alternativa para poder especificar este argumento en milisegundos, cuando cada *Tick* no se produce cada milisegundo, es utilizar la macro *portTICK_PERIOD_MS* del sistema operativo, tal y como se muestra en algún ejemplo de código en el capítulo 2. Una vez enviado el mensaje, se procede a borrar la bandera que indica que se han sobrescritos los datos capturados por los ADCs, que se produce en esta parte del código y que ha sido determinada experimentalmente mediante depuración con JTAG —líneas 14 a la 16—. Luego, se procede a recuperar el mensaje que la aplicación ha mandado —*mensaje_entrante*— a través de la cola *colaDeAplicacion_a_GUI* —líneas de la 18 a la 44—. Al igual que en la emisión del mensaje, en la recepción también se espera un tiempo a recibir el mensaje antes de continuar con la ejecución, en este caso, de 15 milisegundos —línea 18—. Luego, se procede a interpretar el mensaje entrante mediante el bloque *switch-case* entre la líneas 20 a 39. En él, se extraerá el campo tipo de evento del mensaje y dependiendo de su contenido, se ejecutará un caso de código —case—. En este proyecto solo existe un tipo de mensaje que la aplicación mande al entorno gráfico, por tanto, solo existe un caso —tipo de mensaje *DATOS_ESPECTRO*, en línea 22—, se deja la estructura *switch-case* para ampliaciones futuras de mensajes. Una vez determinado que es este el mensaje recibido, se llama a la función del presentador *actualizaSerieFFT*, a través de su interface *modelListener*, pasándole como argumentos los valores de *carga1*, *carga2, carga3* y *carga4* del mensaje entrante. Estos valores significan, respectivamente, el puntero de tipo uint8_t a la región de memoria donde están los datos de la FFT calculados por la aplicación final, el desplazamiento vertical de los datos, la escala vertical de los datos, y canal sobre el que está realizada la FFT. La ejecución de esta función en el presentador se limitará a la ejecución de la función homónima de la vista, pasándole los mismos argumentos. La vista, realmente solo utilizará el primer argumento: la dirección de la región de memoria donde están los datos a representar —el resto de argumentos se deja para posibles ampliaciones futuras—. Los valores de la carga2 y la carga3 del mensaje —desplazamiento vertical y escala vertical—, a pesar de ser mandados en cada mensaje, solo son de interés al ser actualizados debido a una modificación desde la vista en el modelos —de los atributos con repercusión en la aplicación *DesplazamientoVertical* y *EscalaVertical* o *Autopan* y Seguimiento, o el envío a la aplicación del mensaje *AUTOAJUSTA*—. Por ello, en cada recepción del mensaje entrante, se comprueba el campo *TIPO_SUBEVENTO* —líneas de la 29 a la 36— y si este tiene el valor de *ESCALA_VER_DES_VER_MODIFICADOS*, indica que la aplicación ha cambiado el desplazamiento o la escala vertical —o ambos— desde el envío del mensaje anterior. En ese caso, se procede a actualizar las propiedades del modelo —líneas 32 y 33—, y mandar sendos mensajes al presentador —ver sección 4.6.5—, a través de su interface *modelListener* —líneas 35 y 36—. En cualquier caso, ya que la modificación de cualquiera de estos dos valores implica cambios en la determinación del nivel medido, se llama al método del modelo *determinaConversionesNivel* —ver sección 4.6.4— que actualiza las conversiones de nivel. Finalmente, como ya ha sido comentado, se reinicia el siguiente mensaje a mandar con el tipo de evento por defecto —líneas de la 40 a la 43—, *SOLICITUD_DATOS*.

4.6.4 Funciones de utilidad en el modelo: «actualizaTodaLaVista» y «determinaConversionesNivel»

Ya han sido mostradas las propiedades del modelo, sus métodos de acceso y el método *tick* para la comunicación con la aplicación, por un lado, y con el resto del entorno gráfico por otro. Sin embargo, existen dos métodos de utilidad del modelo no descritos aún: el método *actualizaTodaLaVista* y *determinaConversionesNivel*. El primero de ellos sirve, mediante su sola llamada, para que se actualicen todos los controles de la vista de acuerdo a los atributos guardados en el modelo —muy útil en el cambio entre pares vista-presentador—. El segundo ya ha sido mencionado en la sección anterior y sirve para actualizar los parámetros de conversión de nivel, cuando se produce algún cambio en la aplicación relativa al desplazamiento o la escala vertical —atributos en la aplicación final que hacen referencia al valor añadido a los datos y al factor multiplicativo de los mismos, respectivamente, para ajustarse al rango del dato uint8_t—.

El primer método —*actualizaTodaLaVista* — está definido por el siguiente código:

```
1  void Model::actualizaTodaLaVista(void)
2  {
3      modelListener->interpretePresentador(ACTUALIZA_PROP_VENTANA);
4      modelListener->interpretePresentador(ACTUALIZA_PROP_FREC_MUESTREO);
5      modelListener->interpretePresentador(ACTUALIZA_PROP_GAMAS);
6      modelListener->interpretePresentador(ACTUALIZA_PROP_LOGARITMO);
7      modelListener->interpretePresentador(ACTUALIZA_PROP_CANAL);
8      modelListener->interpretePresentador(ACTUALIZA_PROP_ESCALA_HOR);
9      modelListener->interpretePresentador(ACTUALIZA_PROP_DESP_HOR);
10     modelListener->interpretePresentador(ACTUALIZA_PROP_ESCALA_VERT);
11     modelListener->interpretePresentador(ACTUALIZA_PROP_DESPLA_VERT);
12 }
```

Código 4.149: Método del modelo para actualizar toda la vista

Básicamente, instruye al presentador a actualizar en la vista todos aquellos controles relacionados con los valores de los atributos del modelo. Para ello, envía al presentador —concretamente a su método *interpretePresentador*— los mensajes necesarios para que se produzca la actualización. Estos mensajes consisten en unas constantes que el presentador interpreta para obtener el valor del modelo correspondiente y modificar, con él, los controles de la vista. Así, el mensaje —la constante— *ACTUALIZA_PROP_VENTANA*, hace que el presentados solicite al modelo el valor de su atributo *ventana*, mediante su método de lectura, y a continuación actualiza los controles *botonesVentanas[ventana]*, del menú «Ventanas» y *BotonesEstado[0]* del menú «Rápido». El mecanismo concreto del intérprete del presentador será expuesto en la sección 4.6.5. Este método es llamado al final del método setupScreen de la vista:

```
1  void FftView::setupScreen()
2  {
3  ...
4  //*************************************************************************
5  //Se actualiza toda la vista conforme a los atributos del modelo
6  //*************************************************************************
7  presenter->obtenerModelo()->actualizaTodaLaVista();
8  }
```

Código 4.150: Utilización del método actualizaTodaLaVista

El segundo método de utilidad —*determinaConversionesNivel*— tiene el siguiente código:

```
1  void Model::determinaConversionesNivel(void)
2  {
3  if(EscalaLogaritmica)
4  {
5          nivelMenorEnPantalla =20*(DesplazamientoVertical/EscalaVertical)\
6                                  -60.206f+9.542f-48.1308f-20*log10(CG);
7          nivelMayorEnPantalla =20*(255.0f/EscalaVertical)+nivelMenorEnPantalla;
8          nivelEjeVertical = nivelMayorEnPantalla-nivelMenorEnPantalla;
9          nivelPorPixelFFT=nivelEjeVertical/(272.0f*2/3);
10         nivelPorDivision=nivelEjeVertical/4.0f;
11 }
12
13 else
14 {
15         nivelMenorEnPantalla=((DesplazamientoVertical/(EscalaVertical*1024.0f*CG)))\
16                                                                 *(3.0f/255.0f);
17         nivelMayorEnPantalla=(((3.0f/255.0f)*255.0f)/(EscalaVertical*1024.0f*CG))\
18                                                             +nivelMenorEnPantalla;
19         nivelEjeVertical= nivelMayorEnPantalla-nivelMenorEnPantalla;
20         nivelPorPixelFFT=nivelEjeVertical/(272.0f*2/3);
21         nivelPorDivision=nivelEjeVertical/4.0f;
22 }
23 }
```

Código 4.151: Método del modelo para actualizar las conversiones de nivel

Su cometido es determinar qué nivel mínimo y máximo hay en pantalla, qué nivel hay por división vertical de la retícula, y qué factor hay que aplicar para pasar de pixeles a unidades de nivel. Todo esto se hace tanto para unidades lineales —voltios—, como para logarítmicas —dB y dBV—. Por ello, el código se divide en dos bloques de sentencias condicionales, uno para las unidades logarítmicas —líneas de la 3 a la 11— y otro para las lineales —de la línea 13 a la 22—. Se empezará describiendo las conversiones de unidades lineales.

Cuando se realiza el cálculo de la FFT, por parte de la aplicación —objeto *DispositivoFFT*, ver sección 4.5—, se aplica a los datos una corrección cuya finalidad es la de acomodarlos al rango del tipo entero de ocho bits, ya que ese es el tipo de los datos que se va a manejar en el control que represente los datos en el entorno gráfico —objeto de tipo *Graph*, ver sección 4.4.1.11—. Esta corrección es la siguiente:

$$\text{Datos}_{\text{cor\_8b}}[i] = \frac{Datos[i] - Dato_{MIN}}{Dato_{MAX} - Dato_{MIN}} \cdot 255$$

$$\text{Datos}_{\text{cor\_8b}}[i] = \frac{255}{Dato_{MAX} - Dato_{MIN}} \cdot Datos[i] - \frac{255}{Dato_{MAX} - Dato_{MIN}} \cdot Dato_{MIN} \qquad (4.61)$$

$$\text{Datos}_{\text{cor\_8b}}[i] = EscalVer \cdot Datos[i] - DesVer$$

Es decir, a los datos procesados —*Datos[i]*—, se les multiplica por un factor —*EscalVer*—, que se le denomina escala vertical, y al resultado se le resta un valor —*DesVer*—, que se le denomina desplazamiento vertical, para obtener los datos escalados en el rango de un entero de 8 bits —$Datos_{cor\_8b}[i]$—. Como ya ha sido mencionado, la escala vertical y el desplazamiento vertical, son atributos del objeto de la aplicación. El modelo también mantiene estos dos valores en atributos reflejo de los de la aplicación, para tener acceso a ellos de

forma más rápida. De la expresión anterior se pueden deducir el dato mínimo y máximo que se van a representar:

$$Dato_{MIN} = \frac{DesVer}{EscalVer}$$
$$Dato_{MAX} = \frac{255 + Dato_{MIN} \cdot EscalVer}{EscalVer} = \frac{255}{EscalVer} + Dato_{MIN} \quad (4.62)$$

El dato mínimo se obtiene directamente de los parámetros de escala y desplazamiento vertical, y el dato máximo, a partir del dato mínimo y la escala vertical. Para obtener el valor mínimo en tensión, a la expresión del dato mínimo hay que aplicarle el factor de corrección 3/255 —factor para pasar de cuentas del ADC a Voltios—, aplicarle el factor 1/1024, debido a la naturaleza propia de la FFT —ver sección 4.5.3— y el factor 1/CG que corrige la pérdida de amplitud por el uso de la ventana temporal. Por tanto, el valor del dato mínimo, en unidades lineales, se determina como:

$$Dato_{MIN\_V} = \frac{DesVer}{EscalVer \cdot 1024 \cdot CG} \cdot \frac{3}{255}[V] \quad (4.63)$$

Donde $Dato_{MIN\_V}$, es el dato mínimo expresado en voltio y *CG* es la «ganancia coherente» —ver sección 4.5.3—. Esta expresión es la que se codifica en las líneas 15 y 16 del código anterior. Por su parte, para obtener el dato máximo en unidades lineales —Voltio—, se ha de aplicar al dato máximo las mismas correcciones que al dato mínimo:

$$Dato_{MAX\_V} = \left(\frac{255}{EscalVer} + Dato_{MIN}\right)\frac{3}{255 \cdot 1024 \cdot CG} = \frac{3}{255} \cdot \frac{255}{EscalVer \cdot 1024 \cdot CG} + Dato_{MIN\_V} \rightarrow$$
$$Dato_{MAX\_V} = \frac{3}{EscalVer \cdot 1024 \cdot CG} + Dato_{MIN\_V}\ [V] \quad (4.64)$$

Esta expresión se codifica en las líneas 17 y 18 del código anterior. A continuación, se procede a determinar cuánto mide el eje vertical en unidades lineales, consistiendo en la resta entre el valor máximo y el mínimo, previamente calculado. Esto se codifica en la línea 19. Para determinar qué factor hay que aplicar a la cantidad de pixeles para obtener la tensión de dicha cantidad, se divide el eje vertical por el número de pixeles que contiene, en este caso es de 272x2/3 —aplicando función suelo—, ya que esta es la medida vertical de la gráfica. Este cálculo esta codificado en la línea 20. Finalmente, en las conversiones a unidades lineales, se calcula cuánta tensión es cada división vertical de la retícula de la gráfica. Este cálculo es similar al anterior. Basta con dividir el eje vertical por las divisiones verticales, en este caso, cuatro. Esto es codificado en la línea 21.

Para la determinación de las unidades logarítmicas el cálculo es similar, pero tiene cierto grado de complicación añadido. Cuando a la aplicación —objeto de tipo *DsipositivoFFT*— llega la orden de expresar los datos en unidades logarítmicas, esta aplica el logaritmo de base diez a todos los datos, sin multiplicarlos por 20, ya que este es un factor lineal que puede ser aplicado, exclusivamente, al eje vertical, ahorrándose una gran cantidad de operaciones de multiplicación. Luego, se aplica la escala y desplazamiento vertical, sobre los datos

logarítmicos, para ajustarlos al rango del tipo entero de ocho bits, al igual que se hacía con las unidades lineales. Por tanto, la expresión de este ajuste es idéntica al de las unidades lineales:

$$\text{Datos}_{\text{log\_cor\_8b}}[i] = EscalVer_{log} \cdot Datos_{log}[i] - DesVer_{log} \tag{4.65}$$

Aparecen todos los elementos de la expresión anterior con el subíndice «*log*», denotando que se trata de cantidades a las que se le ha aplicado el logaritmo de base diez —$Datos_{log\_cor\_8b}[i]$ y $Datos_{log}[i]$—, o que han sido calculados a partir de cantidades a las que se les ha aplicado —$EscalVer_{log}$ y $DesVer_{log}$—. Para el caso de las unidades logarítmicas, se necesita determinar las cantidades naturales, a partir de las logarítmicas —ya que son de las que se disponen— para poder aplicarles las correcciones. Así, la cantidad del dato mínimo natural, a partir del logarítmico es:

$$Dato_{MIN} = 10^{Dato_{MIN\_log}} \tag{4.66}$$

A partir de esta expresión, se puede calcular el dato mínimo en tensión:

$$\begin{gathered} Dato_{MIN\_V} = \frac{Dato_{MIN}}{1024 \cdot CG} \cdot \frac{3}{255} \rightarrow \\ Dato_{MIN\_V} = \frac{10^{Dato_{MIN\_log}}}{1024 \cdot CG} \cdot \frac{3}{255} \end{gathered} \tag{4.67}$$

Sin embargo, no interesa el dato natural sino el logarítmico, con lo que se aplica 20 por logaritmo a cada lado de la igualdad:

$$\begin{gathered} 20 \cdot log\left(Dato_{MIN\_V}\right) = 20 \cdot log\left(\frac{10^{Dato_{MIN\_log}}}{1024 \cdot CG} \cdot \frac{3}{255}\right) \rightarrow \\ 20log\left(Dato_{MIN\_V}\right) = 20log(10^{Dato_{MIN\_log}}) + 20log(3) - 20log(1024) - 20log(255) - 20log(CG) \\ 20log\left(Dato_{MIM\_V}\right) = 20Dato_{MIN\_log} + 9{,}5424 - 60{,}206 - 48{,}1308 - 20log(CG) \end{gathered} \tag{4.68}$$

Y como similar al caso lineal:

$$Dato_{MIN\_log} = \frac{DesVer_{log}}{EscalVer_{log}} \tag{4.69}$$

Se tiene que el valor mínimo en dBV, a partir de cantidades conocidas, es:

$$20log\left(Dato_{MIM\_V}\right) = 20\,\frac{DesVer_{log}}{EscalVer_{log}} + 9{,}5424 - 60{,}206 - 48{,}1308 - 20log(CG) \tag{4.70}$$

Expresión que se codifica en las líneas 5 y 6. Al igual que para el dato mínimo, para el máximo es necesario determinar su valor natural, a partir del logarítmico, que es:

$$Dato_{MAX} = 10^{Dato_{MAX\_log}} \tag{4.71}$$

Y una vez calculado, aplicar las correcciones para hallar el dato máximo en tensión:

$$Dato_{MAX\_V} = \frac{10^{Dato_{MAX\_log}}}{1024 \cdot CG} \cdot \frac{3}{255} \tag{4.72}$$

Aplicando 20 por logaritmo a los dos lados de la igualdad:

$$20 \cdot log(Dato_{MAX}) = 20 \cdot log\left(\frac{10^{Dato_{MAX\_log}}}{1024 \cdot CG} \cdot \frac{3}{255}\right) \rightarrow$$
$$20log\left(Dato_{MAX\_V}\right) = 20Dato_{MAX\_log} + 9{,}5424 - 60{,}206 - 48{,}1308 - 20log(CG) \tag{4.73}$$

Sustituyendo el valor de $Dato_{MAX\_log}$ en función de la escala vertical y el dato mínimo, se tiene:

$$20log\left(Dato_{MAX\_V}\right) = 20\left(\frac{255}{EscalVer_{log}} + Dato_{MIN\_log}\right) + 9{,}5424 - 60{,}206 - 48{,}1308 - 20log(CG) \tag{4.74}$$

Hay que recordar que $Dato_{MIN\_log}$ es el dato mínimo al que se le ha aplicado un el logaritmo de base 10, que junto con el factor 20 —que lo multiplica fuera del paréntesis— y el resto de sumandos fuera del paréntesis, constituyen el dato mínimo en dBV —$20log(Dato_{MIN\_V})$—. Por tanto, el valor máximo en dBV queda finalmente como:

$$20log\left(Dato_{MAX\_V}\right) = 20 \cdot \frac{255}{EscalVer_{log}} + 20log\left(Dato_{MIM\_V}\right) \tag{4.75}$$

Esta expresión es la que se codifica en la línea 7 del código del método *determinaConversionesNivel*. Lo siguiente, es calcular cuánto mide el eje vertical en dB —línea 19—, y al igual que en el caso de unidades lineales, se calcula como la diferencia entre el nivel mayor menos el menor. De la misma manera, al caso lineal, se calcula la conversión de pixeles a dBV —línea 20—, y la conversión de división vertical de

4.6.5 El Presentador: estructura y función

Tal y como ha sido diseñada la estructura concreta del patrón MVC —modificaciones practicadas en el patrón, ver sección 4.6.1—, las funciones del presentador se reducen a dos: actualizar los controles de la vista, de acuerdo con las modificaciones de los atributos del modelo —realizadas por la vista—, y retransmitir a la vista los datos provenientes de la aplicación final —que vienen a través del modelo—, para que la vista represente estos datos. Por ello, la estructura de la clase presentador se limita, básicamente, a la implementación de sendos métodos que abarquen la funcionalidad descrita. La definición de las partes importantes de esta clase se muestra en la siguiente porción de código:

```
1  class FftPresenter : public Presenter, public ModelListener
2  {
3    public:
4
5      FftPresenter(FftView& v);
6
7      virtual Model* obtenerModelo(void){return model;}
8
9      virtual void actualizaSerieFFT(unsigned char *registroFft,
10                                    float minimo,
11                                    float maximo,
12                                    unsigned char canal);
13     virtual void interpretePresentador(ComandosPresentador comando);
14
15   private:
16
17     FftView& view;
18
19     void actualizaTextosNivelPantalla(void);
20     void actualizaTextosFrecuenciaPantalla(void);
21     char cad_inter1[40];
22     float intermedio;
23 };
```

Código 4.152: Definición de la clase del presentador

La clase concreta del presentador es *FftPresenter*, que deriva de la clase genérica *Presenter* de *TouchGFX* y del interface *ModelListener* —línea 1—. Este interface será comentado al final de esta sección. El método público *obtenerModelo* —línea 7— ya fue comentado el explicar las modificaciones realizadas sobre el patrón *MVC* en la sección 4.6.1. Recordar, aún así, que su finalidad era la de facilitar a la vista la dirección del modelo —que por defecto solo la posee el presentador—, para que de esta forma la vista accediera directamente al modelo y fuera ella la que modificase los atributos de este. Luego están los métodos que realmente dan funcionalidad al presentador. El primero es *actualizaSerieFFT*, encargado de retransmitir a la vista los datos provenientes de la aplicación final que vienen a través del modelo. Como argumentos tiene, por este orden, la dirección de memoria donde están los datos a representar, el valor mínimo y máximo de ellos, y el canal de procedencia de los datos —que puede ser 1 o 2—. El segundo es *interpretePresentador*, que asegura que el presentador actualice los controles de la vista de acuerdo con las modificaciones que la vista hace sobre el modelo. Como elementos privados posee dos métodos para la actualización de textos en pantalla —líneas 22 y 23—, que serán comentados en la siguiente sección 4.6.6, y dos atributos, *cad_inter1* —línea 24— e *intermedio* —línea 25— de uso interno de los métodos. El primero de ellos se utiliza como variable auxiliar para albergar de forma temporal cadenas de texto tanto en el método *interpretePresentador* como en los métodos de actualización de textos. El segundo se usa como dato temporal en los métodos de actualización de textos.

Pero lo importante del presentador son estos dos métodos: *interpretePresentador* y *actualizaSerieFFT*. El diagrama de secuencia para *interpretePresentador* es:

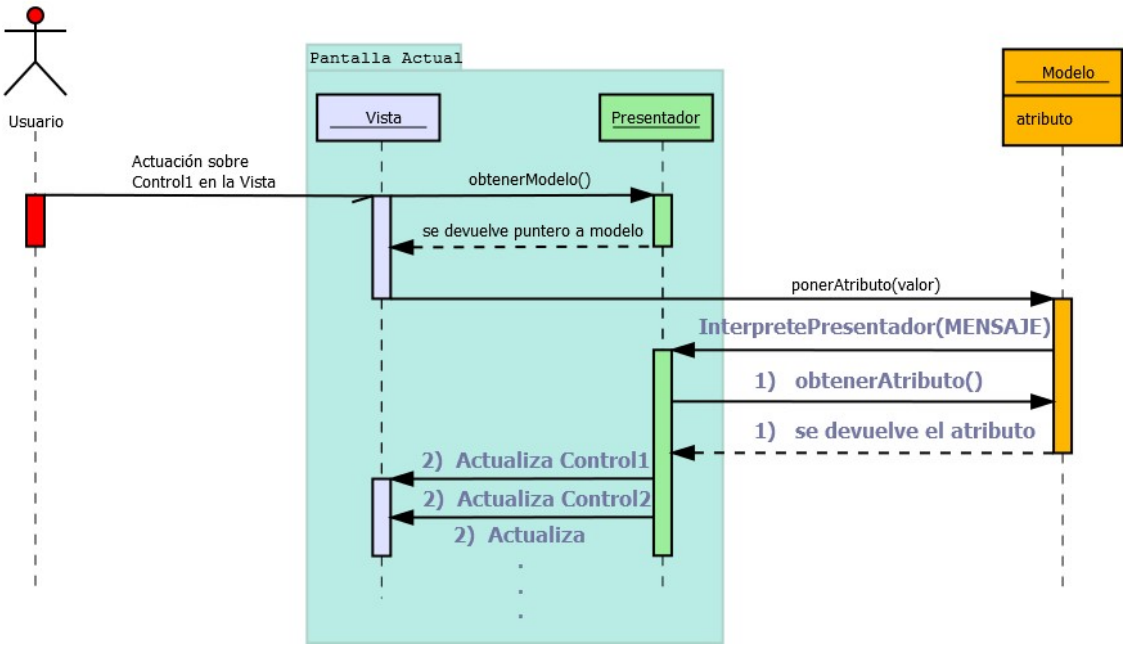


Ilustración 4.66: Diagrama de secuencia en el intérprete del presentador

Todo el proceso se inicia cuando el usuario manipula un control en la vista —en el diagrama aparece como *Control1*— lo que desata que la vista solicite al presentador la dirección del modelo y con ella modifique un atributo en el modelo, relacionado con el control manipulado, a través del método correspondiente de escritura del modelo —en el esquema aparece como *ponerAtributo(valor)*, lo que actualiza el *atributo* del modelo con este valor—. Los pasos de secuencia que ocurren dentro del modelo, para actualizar el atributo, fueron mostrados en el diagrama de secuencia de la sección 4.6.2. A partir de este momento entra en juego el intérprete del presentador. Primeramente el modelo llama al método del presentador *InterpretePresentador* —que constituye el intérprete, más adelante se mostrará su estructura— pasándole como argumento un identificador —una constante— que le indica al presentador qué atributo ha sido modificado en el modelo. El intérprete del presentador realiza, entonces, dos pasos. El primero de ellos consiste en obtener el valor del atributo modificado del modelo. Para ello, el presentador utiliza un método determinado de acceso del modelo —mostrado en el diagrama como 1) *obtenerAtributo()*—. El segundo paso consiste en actualizar el propio control modificado originalmente por el usuario —*Control1*— acorde con el valor del atributo del modelo, así como todos aquellos otros controles de la vista que se vean afectados por el cambio del atributo del modelo —mostrado en el esquema de secuencia con los mensajes precedidos por 2) —. El hecho que el intérprete del presentador modifique el propio control que ha manipulado el usuario y que con dicha manipulación se pueda modificar el estado del control directamente en la vista, sin tener que pasar por todo este proceso, es debido a que el interprete se ha de diseñar de forma que, independiente del origen de la manipulación, realice las modificaciones en todos los controles a los que les afecte el cambio de valor del atributo del modelo. De hecho, el intérprete del presentador no conoce qué control ha provocado la modificación del atributo del modelo.

El intérprete del presentador consiste en un largo bloque de tipo *switch-case* donde se contemplan todos los casos referentes a los posibles mensajes procedentes del modelo. La estructura de este intérprete es la siguiente:

```
1  void FftPresenter::interpretePresentador(ComandosPresentador comando)
2  {
3    // variables temporales para uso en los casos
4    . . .
5    switch (comando)
6     {
7      case ACTUALIZA_PROP_VENTANA:
8        // 1) Recoger el valor(es) de la(s) propiedad(es) del modelo que ha cambiado
9        . . .
10       // 2) Actualizar los widgets que sean afectados por este cambio de propiedad
11       . . .
12       break;
13
14     case ACTUALIZA_PROP_FREC_MUESTREO:
15       // 1) Recoger el valor(es) de la(s) propiedad(es) del modelo que ha cambiado
16       . . .
17       // 2) Actualizar los widgets que sean afectados por este cambio de propiedad
18       . . .
19       break;
20
21     . . .
22    }
23 }
```

Código 4.153: Estructura del intérprete del presentador

El intérprete del presentador recibe un identificador de mensaje —argumento *comando*—, que es pasado al conmutador *switch-case*, realizando en cada caso de interpretación los dos pasos: obtener el valor del atributo que ha originado el mensaje al intérprete, y modificar, consecuentemente, todos los controles de la vista a los que afecte este cambio de valor en el atributo de modelo. Los identificadores de mensajes —que aparecen en las líneas 7 y 14 de los casos presentados, hacen referencia al atributo del modelo que ha sido modificado y que ha disparado el mensaje. Así, *ACTUALIZA_PROP_VENTANA*, significa que ha sido modificado el atributo ventana del modelo o que *ACTUALIZA_PROP_FREC_MUESTRO* ha sido modificado el atributo *muestreador* del modelo. Todos los tipos de mensajes que el modelo puede enviar al intérprete del presentador están recogidos en el archivo de cabecera *interpretePresentador.hpp*, que ha de ser incluido tanto en el archivo de cabecera del modelo como en el archivo de cabecera del presentador. Su contenido es el siguiente:

```
1  #ifndef INTERPRETE_PRESENTADOR_HPP
2  #define INTERPRETE_PRESENTADOR_HPP
3
4  typedef enum TComandosPresentador
5  {
6          ACTUALIZA_PROP_VENTANA,
7          ACTUALIZA_PROP_FREC_MUESTREO,
8          ACTUALIZA_PROP_GAMAS,
9          ACTUALIZA_PROP_AUTOAJUSTE,
10         ACTUALIZA_PROP_SEGUIMIENTO,
11         ACTUALIZA_PROP_LOGARITMO,
12         ACTUALIZA_PROP_CANAL,
13         ACTUALIZA_PROP_ADQUIERE,
14         ACTUALIZA_PROP_DESP_HOR,
15         ACTUALIZA_PROP_ESCALA_HOR,
16         ACTUALIZA_PROP_TEMPORAL,
17         ACTUALIZA_PROP_ESCALA_VERT,
18         ACTUALIZA_PROP_DESPLA_VERT,
19         ACTUALIZA_PROP_AUTOPAN
20 }ComandosPresentador;
21
22 #endif //INTERPRETE_PRESENTADOR_HPP
```

Código 4.154: Archivo de cabecera interpretePresentador.hpp

En él se aprecian los distintos tipos de mensajes que el modelo puede lanzar al presentador.

Un mensaje de especial relevancia es *ACTUALIZA_PROP_DESP_HOR*, ya que su principal función es la de actualizar las conversiones de unidades de frecuencia. La parte de código más importante de este caso del intérprete, que trata este mensaje, es la siguiente:

```
1  case ACTUALIZA_PROP_DESP_HOR:
2   // 1)
3   DespHor=model->obtenerDespHor();
4   EscalaHorizontal=model->obtenerEscalaHorizontal();
5   AnchoPantalla=view.grafico.getWidth();
6   DeltaFrecMuestreo=model->obtenerDeltaFrecMuestreo();
7   nDatos=view.serieFFT.obtennDatos();
8
9   if(EscalaHorizontal<1)
10    IndiceCentral=nDatos/2+DespHor*EscalaHorizontal;
11  else
12    IndiceCentral=nDatos/2+DespHor;
13
14  FrecuenciaCentral = (DeltaFrecMuestreo*IndiceCentral);
15  model->ponerfrecuenciaCentral(FrecuenciaCentral);
16  model->ponerfrecuenciaMenorEnPantalla(FrecuenciaCentral-\
17       ((((AnchoPantalla))/2.0f)*EscalaHorizontal)*DeltaFrecMuestreo);
18  model->ponerfrecuenciaMayorEnPantalla(FrecuenciaCentral\
19       +((((AnchoPantalla))/2.0f-)*EscalaHorizontal)*DeltaFrecMuestreo);
20  model->ponerfrecuenciaPorPixel(DeltaFrecMuestreo *EscalaHorizontal);
21  model->ponerfrecuenciaSpan((AnchoPantalla-1)*model->obtenerfrecuenciaPorPixel());
22  model->ponerfrecuenciaPorDivision(model->obtenerfrecuenciaPorPixel()*(AnchoPantalla/8.0f));
23  . . .
24  break;
```

Código 4.155: Mensaje en el presentador para actualizar el desplazamiento horizontal

Al principio del código se utilizan variables temporales —líneas de la 3 a la 7, que están definidas al inicio del método del intérprete del presentador—, donde se recogen los tributos del modelo y de controles gráficos —concretamente del control «*grafico*», en línea 5 y del control «*serieFFT*» en línea 7—. Luego, se determina el índice central, que se recuerda que es índice del dato del registro de datos —tanto en el dominio del tiempo como en el de la frecuencia, más habitualmente de este último— que se encuentra en la parte media de la pantalla. Este se calcula a partir de la dimensión horizontal de la pantalla —de su longitud media—, del desplazamiento horizontal y de la escala horizontal —ver sección 4.4.1.11— como:

$$\begin{aligned} i_{CENTRAL} &= \frac{n\_Datos}{2} + DespHor \cdot EscalHor, && si\ \ EscalHor < 1 \\ i_{CENTRAL} &= \frac{n\_Datos}{2} + DespHor, && si\ \ EscalHor \geq 1 \end{aligned} \qquad (4.76)$$

Donde $i_{CENTRAL}$ es el índice central, *n_Datos*, el número de datos en el registro —1024—, *DespHor*, es el desplazamiento horizontal, que cuando la escala horizontal es menor a uno, representa desplazamientos de pixeles, incrementándose en unidades e indexando un nuevo dato del registro cuando se producen tantos desplazamientos como indica el valor de la inversa de la escala horizontal, mientras que cuando la escala horizontal es mayor o igual a uno, representa desplazamiento de datos del registro, y se incrementa en el valor de la escala horizontal, produciendo un desplazamiento de un pixel en cada incremento de esta cantidad. *EscalHor* es la escala horizontal, que cuando es mayor a uno, representa cuantos datos consecutivos del registro se representan horizontalmente sobre el mismo pixel, mientras que si es menor, indica cuantos pixeles separan dos datos consecutivos del registro. Estas dos expresiones son codificadas de la línea 9 a la 12 del código mostrado. A partir de ellas se calcula la frecuencia que le corresponde mediante la siguiente expresión:

$$Frecuencia_{CENTRAL} = i_{CENTRAL} \cdot \Delta F \quad (4.77)$$

Donde $Frecuencia_{CENTRAL}$ es la frecuencia que aparece en la mitad de la pantalla, $i_{CENTRAL}$, es el índice central, antes determinado, y ΔF es el incremento de frecuencia de un dato al siguiente en el registro de datos, determinada al modificar la velocidad del muestreador —actualizar el atributo *muestreador* del modelo a través de su método *ponerMuestreo*—. Esta expresión es codificada en la línea 14 del código y actualiza la propiedad *FrecuenciaCentral* del modelo a través de su método de acceso *ponerFrecuenciaCentral* en la línea 15. Luego, se calculan las frecuencias menor y mayor que aparecen en pantalla como:

$$F_{MENOR} = Frecuencia_{CENTRAL} - \frac{Ancho_{Pantalla}}{2} \cdot EscalHor$$
$$F_{MAYOR} = Frecuencia_{CENTRAL} + \frac{Ancho_{Pantalla}}{2} \cdot EscalHor \quad (4\ 78)$$

Donde F_{MENOR}, es la frecuencia menor en pantalla, F_{MAYOR}, la mayor, y $Ancho_{Pantalla}$, la longitud horizontal de la pantalla en pixeles. Estas dos expresiones son codificadas en las líneas de la 16 a la 19, actualizando, respectivamente, las propiedades del modelo *FrecuenciaMenorEnPantalla* y *FrecuenciaMayorEnPantalla*, a través de sus respectivos métodos de escritura. Finalmente se calcula la frecuencia que representa cada pixel, el ancho de frecuencia del eje horizontal y cuanta frecuencia representa cada división horizontal de la retícula, mediante las siguientes expresiones:

$$F_{PIXEL} = \Delta F \cdot EscalHor$$
$$SPAN = (Ancho_{Pantalla} - 1) \cdot F_{PIXEL}$$
$$F_{DIVISION} = \left(\frac{Ancho_{Pantalla}}{8}\right) \cdot F_{PIXEL} \quad (4.79)$$

Donde F_{PIXEL}, es la frecuencia que le corresponde a cada pixel, *SPAN*, es el ancho en frecuencia del eje horizontal y $F_{DIVISION}$, el ancho de frecuencia de cada división horizontal de la retícula. Estas expresiones son codificadas y actualizan los atributos correspondientes del modelo en las líneas 20, 21 y 22 del código anterior.

Para actualizar la serie con los datos procedentes de la aplicación —que es la otra gran función que ha de cumplir el presentador—, el presentador sigue el siguiente diagrama de secuencia:

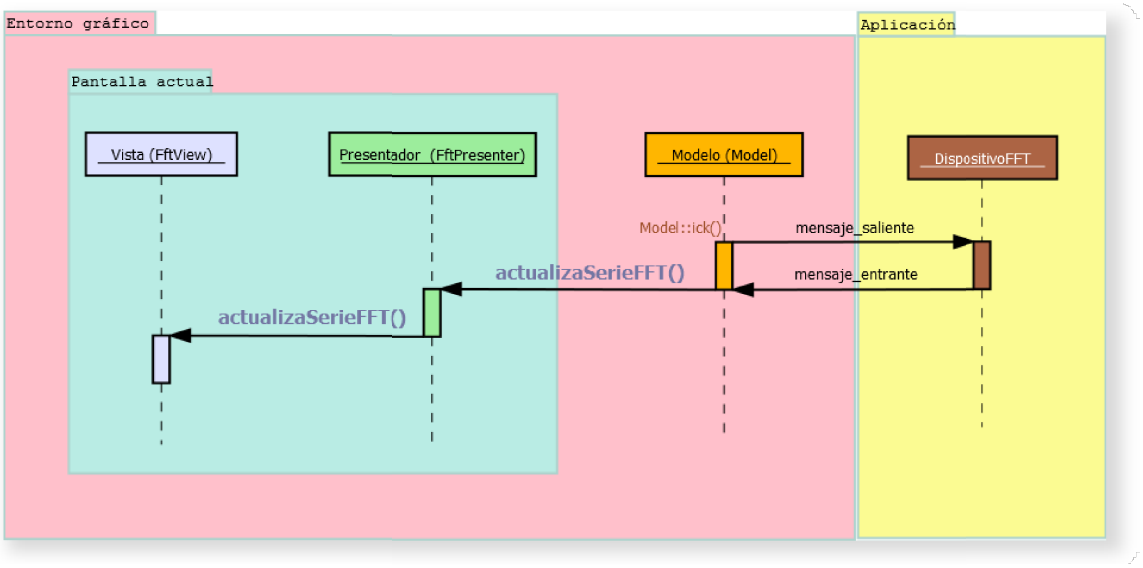


Ilustración 4.67: Secuencia de retransmisión de los datos desde el presentador a la vista

En esta secuencia se observa que al presentador le llegan los datos del registro, procesado por la aplicación, a través del modelo, consistente en la llamada a su método *actualizaSerieFFT* —pasándole como primer argumento la dirección de memoria donde se localiza el registro—. El presentador se limita a llamar a la función homónima de la vista, pasándole los argumentos que recibe del modelo mediante el siguiente código:

```
1  void FftPresenter::actualizaSerieFFT(unsigned char *registroFft,
2                                       float DesplazamientoVertical,
3                                       float EscalaVertical,
4                                       unsigned char canal)
5  {
6    view.actualizaSerieFFT(registroFft, DesplazamientoVertical, EscalaVertical, canal);
7  }
```

Código 4.156: Función de retransmisión de los datos desde el presentador a la vista

En las secuencias mostradas se ha indicado la comunicación entre el modelo y el presentador como directa, sin embargo se realiza indirectamente a través del interface *ModelListener* que el presentador implementa:

```
1  class ModelListener
2  {
3   public:
4      . . .
5      virtual void actualizaSerieFFT(unsigned char *registroFft,
6                                     float minimo,
7                                     float maximo,
8                                     unsigned char canal){}
9
10    virtual void interpretePresentador(ComandosPresentador comando){}
11
12   protected:
13      Model* model;
14 };
```

Código 4.157: Cuerpo de la clase ModelListener utilizada

El presentador hereda desde esta clase, con lo cual tiene acceso al modelo, ya que *ModelListener* posee un puntero al modelo —en su parte protegida, línea 13—. El modelo tiene acceso al presentador, ya que el primero tiene un puntero a *ModelListener*, pero desde su punto de vista, el presentador es de la clase *ModelListener* y por tanto, solo puede acceder en el presentador a lo definido en esta clase, en este caso, a los métodos *actualizaFFT*

e *interpretePresentador*. Como se aprecia en el código, estas dos funciones han sido declaradas pero también definidas con cuerpo vacio, mientras que en archivo fuente del presentador son sobrescritas con el código necesario para darle funcionalidad. Mediante el mecanismo de polimorfismo del lenguaje C++, el modelo al llamar a cualquiera de estas dos funciones de *ModelListener*, llama a la versión definida en el presentador. La única restricción que deben cumplir las funciones definidas en *ModelListener* es que tengan cuerpo vacio. La definición final de estos cuerpos se localizará en el presentador correcto. De esta manera, el modelo tiene que tener siempre la posibilidad de llamar a todas las funciones definidas en todos los presentadores de la aplicación gráfica, aunque solo hay un presentador cargado a un tiempo. Así, si el modelo llama a una función de *ModelListener* que pertenece a un presentador no cargado, llamará a la versión con cuerpo vacio definida en *ModelListener* y no se ejecutará ningún código en el presentador actual, pero si llama a una función de *ModelListener* que pertenece al presentador actualmente en uso, llamará al código de la función definido en este presentador. En el caso de este proyecto, al utilizar solo un par vista-presentador —no hay conmutación entre varias vistas—, todas las funciones definidas con cuerpo vacio de *ModelListener* son las redefinidas con código en el único presentador usado: *FftPresenter*.

4.6.6 Funciones de utilidad en el presentador

Existen dos métodos en el presentador que no son los comentados en secciones anteriores y que ayudan al intérprete del presentador a realizar su labor de actualizar la vista. Estos métodos son: *actualizaTextosNivelPantalla* y *actualizaTextosFrecuenciaPantalla*.

El primero tiene como misión la de actualizar los textos referentes a nivel que aparecen en la vista. Estos textos son: el texto que informa del mayor nivel en pantalla, el texto que informa del menor nivel en pantalla y el texto que informa qué nivel representa cada división vertical de la retícula. Esta función es utilizada por los mensajes mandados por el modelo referentes al nivel: *ACTUALIZA_PROP_ESCALA_VERT*, *ACTUALIZA_PROP_DESPLA_VERT* y *ACTUALIZA_PROP_VENTANA*. La estructura de este método es la siguiente:

```
1  void  FftPresenter::actualizaTextosNivelPantalla(void)
2  {
3    intermedio=model->obtenerNivelMayorEnPantalla();
4    if(model->obtenerEscalaLogaritmica())
5      snprintf (cad_inter1, 20,"%.2f dBV", intermedio);
6    else
7      snprintf (cad_inter1, 20,"%.3f V", intermedio);
8    view.cambiaPuntoPorComa(cad_inter1);
9    Unicode::strncpy(view.cadenaMaxNivel, cad_inter1, 20);
10   view.texMaxNivel.resizeToCurrentText();
11   view.texMaxNivel.setXY(5,3+view.texMaxNivel.getTextHeight());
12   view.texMaxNivel.invalidate();
13
14   //Sentencias similares para actualizar el control textual
15   //texMinNivel que representa el nivel minimo en pantalla
16
17   //Sentencias similares para actualizar el control textual
18   //texNivelDivision que representa el nivel por división vertical en pantalla
19 }
```

Código 4.158: Estructura del método actualizaTextosNivelPantalla del presentador

Para mostrar el máximo nivel en pantalla —en el control gráfico *texMaxNivel*, un control de texto—, inicialmente se recoge, en la variable temporal «*intermedio*» del presentador—línea 3—, el valor del mayor nivel que está guardado en el modelo y que ha sido determinado al llamar al método *determinaConversionesNivel* desde el método *tick* del modelo —ver secciones 4.6.3 y 4.6.4—. Este valor numérico es pasado a cadena entre las líneas 4 a 7, dentro de la variable temporal del presentador *cad_inter1*. Luego, al valor en formato cadena se

le cambia el separador de decimales para que esté en formato de la lengua Castellana —línea 8— mediante la función de utilidad de la vista *cambiaPuntoPorComa*. A continuación la cadena temporal es asignada a la cadena utilizada por el control *texMaxNivel* —línea 9—, ajustando su tamaño al de la cadena —línea 10—, posicionándola —línea 11— y solicitando su refresco —línea 12—. Para los otros dos controles de texto, *texMinNivel* y *texNivelDivision*, las sentencia para actualizarlos, dentro de este mismo método, son similares.

El segundo método actualiza los textos en la vista referentes a cantidades de frecuencia. Estos textos son: el referente a la frecuencia central, el referente a la frecuencia por división, el referente a la frecuencia menor en pantalla y el referente a la mayor. Este método es llamado al gestionar los mensajes *ACTUALIZA_PROP_DESP_HOR*, *ACTUALIZA_PROP_ESCALA_HOR* y *ACTUALIZA_PROP_VENTANA*. Su estructura es la siguiente:

```
1  void FftPresenter::actualizaTextosFrecuenciaPantalla(void)
2  {
3    intermedio=model->obtenerfrecuenciaCentral();
4    snprintf (cad_inter1, 20,"%.3f kHz", intermedio);
5    view.cambiaPuntoPorComa(cad_inter1);
6    Unicode::strncpy(view.cadenaPanFrecCent, cad_inter1, 20);
7    view.texPanFrecCent.resizeToCurrentText();
8    view.texPanFrecCent.setXY(view.graphContainer.getWidth()/2-\
9                              view.texPanFrecCent.getWidth()/2,2);
10   view.texPanFrecCent.invalidate();
11
12   //Sentencias similares para actualizar el control de texto
13   //texFrecDivision que representa la frecuencia por división horizontal
14
15   //Sentencias similares para actualizar el control de texto
16   //texMinFrec que representa la frecuencia mínima en pantalla
17
18   //Sentencias similares para actualizar el control de texto
19   //texMaxFrec que representa la frecuencia máxima en pantalla
20 }
```

Código 4.159: Estructura del método actualizaTextosFrecuenciaPantalla del presentador

Para actualizar el texto en pantalla que muestra la frecuencia central, se recoge el dato guardado en el modelo referente a esta información —y que ha sido calculado al gestionar en el presentador los mansajes *ACTUALIZA_PROP_DESP_HOR* y *ACTUALIZA_PROP_ESCALA_HOR*— en la variable numérica temporal del presentador «*intermedio*» —línea 3—. A continuación es pasado a cadena —línea 4—, sustituyendo el punto decimal por la coma —línea 5—, y pasada a la cadena que utiliza el control de texto —línea 6—. Luego se procede a redimensionar y posicionar el texto para finalmente redibujarlo —líneas de la 7 a la 10—. Procesos similares se usan para los textos que muestran la frecuencia por división, la frecuencia mínima y la máxima.

4.6.7 Distribución de componentes gráficos en la vista

La vista es la entidad del patrón MVC encargado de contener todos los controles gráficos, como también albergar su código de inicialización y configuración, y realizar las primeras actuaciones al producirse los eventos. Por ello, en esta sección se va a mostrar la distribución de controles definidos en el archivo de cabecera de la vista así como una pequeña parte del código contenido en el archivo fuente de la vista necesaria para entender la inicialización de los controles. Para conocer la organización visual y estética de los controles, ver la sección 1.9. Finalmente, se mostrará el código de los métodos *handleTickEvent*, que realiza el autoajuste después de la inicialización de los controles, y *actualizaSerieFFT*, que refresca de forma periódica los dotas en el gráfico.

La definición de controles en el archivo de cabecera de la vista está distribuida de forma que agrupe aquellos que están relacionados. Esta distribución es la siguiente:

```
class FftView : public View<FftPresenter>
{
  public:
    . . .
  private:
    /// Declaración de controles sin agrupación específica

    /// Declaración de controles encima del gráfico

    /// Declaración de elementos de menú

    /// Declaración de controles de cada elemento de menú

    /// Declaración del control que representa el menú
};
```

Código 4.160: Distribución de declaraciones en el archivo FftView.hpp

Como en toda clase, las declaraciones aparecen en la parte privada. Estas, se dividen en aquellas que no están agrupadas, en las que se refieren a controles situados encima del gráfico, en las que constituyen elementos de menú, en las que se refieren a controles en elementos de menú y a la declaración del propio menú.

Los controles no agrupados son los siguientes

```
//*****************************************************************************
//Declaración de controles de fondo
//*****************************************************************************
  Box FondoMenu;
  Box FondoGrafico;

//*****************************************************************************
//Declaración de la rejilla
//*****************************************************************************
  Image rejillaAnterior;
  Image rejillaPosterior;

//*****************************************************************************
//Declaración de controles para el gráfico
//*****************************************************************************
  Graph grafico;
  Graph::Serie serieFFT;
  Container graphContainer;
```

Código 4.161: Declaraciones de controles sin agrupación en la vista

Estos son los que constituyen el fondo del gráfico y el fondo de la región donde se localizan los elementos de menú —líneas 4 y 5, respectivamente—, las dos rejillas que están tanto detrás como delante del gráfico —líneas 10 y 11—, y los elementos que constituyen el conjunto del gráfico, que son : el objeto *grafico* —línea 16—, que representa la gráfica donde se incorporarán los datos; el objeto *serieFFT* —línea 17—, que representa los datos del gráfico; y *graphContainer* —línea 18—, contenedor que contendrá, en este orden, el fondo gráfico, la rejilla anterior, el objeto gráfico, el cursor 1, el cursor 2 y la rejilla posterior.

Encima del conjunto del gráfico —el contenedor del gráfico—, se localizan la siguiente serie de controles:

```
1  //*************************************************************************
2  //Declaración de controles sobre el gráfico
3  //*************************************************************************
4    TBotonOcultado< TPulsador< Button > >botonOcultado_masV;
5    TBotonOcultado< TPulsador< Button > >botonOcultado_menosV;
6    TBotonOcultado< TPulsador< Button > >botonOcultado_masH;
7    TBotonOcultado< TPulsador< Button > >botonOcultado_menosH;
8    TAgrupacionBotonesOcultados<TPulsador< Button > > grupoOcultadoV;
9    TAgrupacionBotonesOcultados<TPulsador< Button > > grupoOcultadoH;
10
11   ClickListener<Marcador> ledActivador;
12   TouchAreaMio areaTocable;
13
14 //*************************************************************************
15 //Declaraciones de textos encima del gráfico
16 //*************************************************************************
17   TextAreaWithOneWildcard texPanFrecCent;
18   Unicode::UnicodeChar cadenaPanFrecCent[25];
19   TextAreaWithOneWildcard texMinNivel;
20   Unicode::UnicodeChar cadenaMinNivel[15];
21   TextAreaWithOneWildcard texMaxNivel;
22   Unicode::UnicodeChar cadenaMaxNivel[15];
23   TextAreaWithOneWildcard texNivelDivision;
24   Unicode::UnicodeChar cadenaNivelDivision[15];
25   TextAreaWithOneWildcard texFrecDivision;
26   Unicode::UnicodeChar cadenaFrecDivision[15];
27
28   TextAreaWithOneWildcard texMinFrec;
29   Unicode::UnicodeChar cadenaMinFrec[15];
30   TextAreaWithOneWildcard texMaxFrec;
31   Unicode::UnicodeChar cadenaMaxFrec[15];
```

Código 4.162: Declaración de controles encima del gráfico en la vista

Estos controles son los cuatro botones que de ocultación —líneas de la 4 a la 9— que controlan el escalado de los ejes del gráfico; el área invisible donde descansan estos botones —línea 12—; el control que indica si el sistema está o no adquiriendo —línea 11—; y los controles de texto que informan al usuario de datos relacionados con información de nivel y frecuencia.

A continuación están los elementos de menú:

```
1  //*******************************************************
2  //Declaración de menús
3  //*******************************************************
4    ElementoMenu elementoMenu0;
5    ElementoMenu elementoMenu1;
6    ElementoMenu elementoMenu2;
7    ElementoMenu elementoMenu3;
8    ElementoMenu elementoMenu4;
9    ElementoMenu elementoMenu5;
10   ElementoMenu elementoMenu6;
```

Código 4.163: Declaraciones de elementos de texto en la vista

Hay en total siete elementos de menú, que tal y como fueron mostrados en la sección 1.9 son: *Rápido*, *Controles*, *Ventanas*, *Esquemas*, *Pantalla*, *Cursores* y *Adquisición*. Sin embargo, a pesar de ser este el orden gráfico, este no es el orden en que aparecen en el código anterior. Así *elementoMenu0* es el menú *Rápido*,

elementoMenu1 es el menú *Esquemas*, *elementoMenu2* es el menú *Adquisición*, *elementoMenu3* es el menú *Cursores*, *elementoMenu4* es el menú *Ventanas*, *elementoMenu5* es el menú Controles y *elementoMenu6* es el menú *Pantalla*.

Los siguientes objetos declarados son los pertenecientes al elemento de menú *Rápido*:

```
1  //*******************************************************
2  //Declaración de controles del menu "RAPIDO"
3  //*******************************************************
4    #define N_BOTONES_ESTADO 7
5    BotonEstados BotonesEstado[N_BOTONES_ESTADO];
6    ListLayout lista;
7    ScrollableContainer contiene;
8    Box recuadro;
9    Marcador marcadorIzq;
10   Marcador marcadorDer;
```

Código 4.164: Declaraciones de objetos del menú Rápido de la vista

En este menú se localizan los botones de estado que engloban la mayor parte de la funcionalidad del resto de controles, haciendo de este menú un menú de acceso «rápido». Los botones de estado se crean en el *array BotonesEstado* —línea 5— de siete elementos. Estos botones van dentro del contenedor *Lista* —línea 6— para que tengan ordenamiento gráfico horizontal, y a su vez, este está contenido en el contenedor *contiene* —línea 7— para dar a todo el conjunto de botones la funcionalidad de deslizamiento —*scroll*—. El conjunto de botones descansa sobre un rectángulo traslúcido —línea 8—, y a ambos lados del mismo se localizan dos flechas que parpadean, indicando que el conjunto se puede deslizar a ambos lados —objetos de tipo Marcador; líneas 9 y 10—. El siguiente conjunto de controles declarados son los pertenecientes al elemento de menú *Controles*:

```
1  //*******************************************************************************
2  //Declaración de botones de estado del menu "CONTROLES"
3  //*******************************************************************************
4    Container contContr1, contContr2, contContr3, contContr4;
5    SwipeContainer contControles;
6    //-------------------------------------------------------
7    HoldableButton decFrecCentral;
8    HoldableButton incFrecCentral;
9    TextArea textFrecCentral;
10   TextAreaWithOneWildcard textoDesHor;
11   Unicode::UnicodeChar cadenaDesHor[20];
12   TextAreaWithOneWildcard textoDesHor2;
13   Unicode::UnicodeChar cadenaDesHor2[20];
14   //-------------------------------------------------------
15   Button decSpan;
16   Button incSpan;
17   TextArea textSpan;
18   TextAreaWithOneWildcard textoEscHor;
19   Unicode::UnicodeChar cadenaEscHor[20];
20   TextAreaWithOneWildcard textoEscHor2;
21   Unicode::UnicodeChar cadenaEscHor2[20];
22   TextAreaWithOneWildcard textoDesHorMax;
23   Unicode::UnicodeChar cadenaDesHorMax[8];
24   TextAreaWithOneWildcard textoDesHorMin;
25   Unicode::UnicodeChar cadenaDesHorMin[8];
26   //-------------------------------------------------------
27   HoldableButton decDesplazamiento;
28   HoldableButton incDesplazamiento;
29   TextArea textDesplazamiento;
30   TextAreaWithOneWildcard textoDesVert;
31   Unicode::UnicodeChar cadenaDesVert[20];
32   //-------------------------------------------------------
33   Button decAmplitud;
34   Button incAmplitud;
35   TextArea textAmplitud;
36   TextAreaWithOneWildcard textoEscVert;
37   Unicode::UnicodeChar cadenaEscVert[20];
```

Código 4.165: Declaraciones de objetos del menú Controles de la vista

Este menú consta de un contenedor deslizable —línea 5— con cuatro vistas, cada una de las cuales está ocupada por los contenedores declarados en la línea 4. La primera vista es referente a los objetos destinados a controlar la frecuencia central —líneas de la 7 a la 13—. La segunda, a controlar el ancho de frecuencia del eje horizontal de la gráfica —líneas de la 15 a la 25—. La tercera, a controlar la posición vertical de la gráfica —líneas de la 27 a la 31—. Y finalmente, la cuarta, a controlar la escala vertical —líneas de la 33 a la 37—.

El siguiente grupo de controles declarados es el perteneciente al elemento de menú *Ventanas*:

```
1  //************************************************************
2  //Declaración de controles del menu "VENTANAS"
3  //************************************************************
4    static const uint16_t NUMERO_BOTONES_VENTANAS = 5;
5    RadioButtonGroup<NUMERO_BOTONES_VENTANAS> grupoBotonesVentanas;
6    RadioButton botonesVentanas[NUMERO_BOTONES_VENTANAS];
7    TextArea textRectangular;
8    TextArea textHanning;
9    TextArea textHamming;
10   TextArea textBlacmanHarris;
11   TextArea textBarlett;
12   ButtonWithLabel botAutoajuste;
```

Código 4.166: Declaraciones de objetos del menú Ventanas de la vista

En este grupo se declaran los cinco botones de opción, mediante el *array botonesVentanas* —línea 6—, que controlan la aplicación de cada ventana temporal, así como el grupo lógico que los engloban —línea 5—. También se declaran los textos que van dentro de cada botón de opción —líneas de la 7 a la 11— así como el botón de autoajuse —línea 12—.

Los siguientes objetos declarados son los pertenecientes al elemento de menú *Esquemas*:

```
1  //*******************************************************
2  //Declaración de controles del menu "ESQUEMAS"
3  //*******************************************************
4    Image imagenesGamas[N_GAMAS];
5    RadioButtonGroup<N_GAMAS> grupoBotonesGamas;
6    RadioButton botonesGamas[N_GAMAS];
7    RadioButton botonActivacionGamas;
8    TextArea texActivar;
```

Código 4.167: Declaraciones de objetos del menú Esquemas de la vista

En este menú se declaran siete botones de opción —línea 6—, su grupo lógico —línea 5—, así como siete imágenes —línea 4—, relativos todos ellos a los siete posibles esquemas de color a utilizar en la gráfica. También se declaran el botón de activación de los esquemas —línea 7— y su texto interno —línea 8—.

En el siguiente grupo, se declaran los controles contenidos en el elemento de menú *Pantalla*:

```
1  //************************************************************
2  //Declaración de controles del menu "PANTALLA"
3  //************************************************************
4    Image pantallaFondo1, pantallaFondo2, pantallaFondo3;
5    Container contPant1, contPant2, contPant3;
6    SwipeContainer contPantallas;
7    //----------------------------------------------------------
8    TextArea texTraza;
9    TextArea texVectoresTraza;
10   TextArea texColorTraza;
11   TextAreaWithOneWildcard texTamTraza;
12   Unicode::UnicodeChar cadenatexTamTraza[10];
13   RadioButton botonActivacionVectores;
14   SliderBar deslizadorColorTraza;
15   SliderBar deslizadorAnchoTraza;
16   //----------------------------------------------------------
17   TextArea texRejilla;
18   TextArea texTraseraRejilla;
19   TextArea texVisibleRejilla;
20   TextArea texTColorFondo;
21   RadioButton botonPosicionRejilla;
22   RadioButton botonVisibleRejilla;
23   SliderBar deslizadorColorFondo;
24   SliderBar deslizadorLuminanciaFondo;
25   //----------------------------------------------------------
26   TextArea texTextos;
27   TextArea texFrecuenciaTexto;
28   TextArea texNivelTexto;
29   RadioButton botonTextosFrecuancia;
30   RadioButton botonTextosNivel;
31   SliderBar deslizadorColorTextoFrec;
32   SliderBar deslizadorColorTextoNiv;
```

Código 4.168: Declaraciones de objetos del menú Pantalla de la vista

En este menú se declaran los objetos con una estructura similar al menú *Controles*. Igualmente se utiliza un contenedor deslizable —línea 6— con tres contenedores ordinarios —línea 5— y sus correspondientes imágenes de fondo —línea 4—. En el primer contenedor se declaran los objetos necesarios para controlar el color y anchura de la traza de la gráfica y su representación en forma de puntos o vectores—líneas de la 9 a la 15—; en el segundo, los objetos que controlan la posición de la retícula y el color e iluminación del fondo del gráfico —líneas de la 17 a la 24—; en el tercero, los objetos que controlan la visibilidad de los textos en pantalla, así como su color —líneas de la 26 a la 32—.

El siguiente elemento de menú donde se declaran objetos, es el elemento de menú *Cursores*:

```
1  //*****************************************************************
2  //Declaración de controles del menu "CURSORES"
3  //*****************************************************************
4    Container contCur1, contCur2, contCur3;
5    SwipeContainer contCursores;
6    //-------------------------------------------------------
7    Cursor Cursor1;
8    Cursor Cursor2;
9    JogWheel RuedaCursor1;
10   JogWheel RuedaCursor2;
11   Box CuadroInformacionCursores;
12
13   TextArea texTituloCursor1;
14   TextArea texTituloCursor2;
15   TextArea texTituloX;
16   TextArea texTituloY;
17   TextArea textTituloDelta;
18
19   TextAreaWithOneWildcard texXCursor1;
20   Unicode::UnicodeChar cadenaXCursor1[15];
21   TextAreaWithOneWildcard texYCursor1;
22   Unicode::UnicodeChar cadenaYCursor1[15];
23   TextAreaWithOneWildcard texXCursor2;
24   Unicode::UnicodeChar cadenaXCursor2[15];
25   TextAreaWithOneWildcard texYCursor2;
26   Unicode::UnicodeChar cadenaYCursor2[15];
27   TextAreaWithOneWildcard texDeltaX;
28   Unicode::UnicodeChar cadenaDeltaX[15];
29   TextAreaWithOneWildcard texDeltaY;
30   Unicode::UnicodeChar cadenaDeltaY[15];
31   //-------------------------------------------------------
32   TextArea texContenedorCursor1;
33   TextArea texColorCursor1;
34   TextArea texLuminanciaCursor1;
35   TextArea texAlfaCursor1;
36   SliderBar deslizadorColorCursor1;
37   SliderBar deslizadorLuminanciaCursor1;
38   SliderBar deslizadorAlfaCursor1;
39   //-------------------------------------------------------
40   TextArea texContenedorCursor2;
41   TextArea texColorCursor2;
42   TextArea texLuminanciaCursor2;
43   TextArea texAlfaCursor2;
44   SliderBar deslizadorColorCursor2;
45   SliderBar deslizadorLuminanciaCursor2;
46   SliderBar deslizadorAlfaCursor2;
```

Código 4.169: Declaraciones de objetos del menú Cursores de la vista

De forma similar al menú *Pantalla* y *Controles*, en el menú *Cursores* se declara un contenedor deslizable —línea 5—, con tres contenedores ordinarios —línea 4—. En el primer contendor se declaran los objetos que controlan los dos cursores así como los textos que muestran los datos de estos —líneas de la 7 a la 30—. En el segundo contenedor se declaran los objetos que controlan la apariencia visual de primer cursor como es su color, su luminosidad y su transparencia —líneas de la 32 a la 38—. En el tercer contenedor se hace lo propio para el segundo cursor —líneas de la 40 a la 46—.

Las declaraciones de objetos del último elemento de menú —Adquisición—, se realiza a continuación:

```
1  //************************************************************************
2  //Declaración de controles del menu "ADQUI."
3  //************************************************************************
4    Image imgSeparador1;
5    Image imgSeparador2;
6    Image imgFondoSelector;
7
8    TextArea texAdquisicion;
9    RadioButton botonActivacionAdquisicion;
10
11   TextArea texTemporal;
12   RadioButton botonActivacionTemporal;
13
14   TextArea texCanal1;
15   TextArea texCanal2;
16   RadioButton botonCanal1;
17   RadioButton botonCanal2;
18   RadioButtonGroup<2> grupoBotonesCanales;
19
20   TextArea texLogaritmo;
21   RadioButton botonActivacionLogaritmo;
22
23   TextArea texAutoPan;
24   RadioButton botonActivacionAutoPan;
25
26   ElementoSelector elementosFs[4];
27   Selector<4> contenedorElementosFs;
```

Código 4.170: Declaraciones de objetos del menú Adquisición de la vista

En este menú se declaran los botones de opción que controlan la adquisición —línea 9—, la representación temporal —línea 12—, la selección del los canales —líneas de la 16 a la 18—, el botón de opción de representación logarítmica —línea 21—, el botón de opción de la activación del *autopan* —línea 24— y los cuatro elementos de selector —línea 26— que permiten cambiar la velocidad de muestreo entre la cuatro posibles.

Finalmente, en la declaración de objetos, se declara el menú donde están contenidos todos los menús anteriores —elementos de menú—:

```
1  //************************************************
2  //Declaración del menú principal
3  //************************************************
4    Menu menu1;
```

Código 4.171: Declaraciones del menú principal de la vista

El código en el que se produce la agregación de cada elemento de menú al menú, y este y otros controles a la vista, es el archivo fuente de la vista —FftView.ccp—. En la siguiente porción de código, se muestran las sentencias que hacen esto en el archivo fuente de la vista:

```
//*****************************************************************************
//Agregación de controles al menú principal y configuración del mismo
//*****************************************************************************
  menu1.agrega(elementoMenu2);//ADQUI.
  menu1.agrega(elementoMenu3);//CURSORES
  menu1.agrega(elementoMenu6);//PANTALLA
  menu1.agrega(elementoMenu1);//ESQUEMAS
  menu1.agrega(elementoMenu4);//VENTANAS
  menu1.agrega(elementoMenu5);//CONTROLES
  menu1.agrega(elementoMenu0);//RAPIDO
  menu1.ponerInicioMenu((menu1.getWidth()-\
        (menu1.obtenN_Elementos())*elementoMenu0.getWidth())/2);//pone el menú centrado
  menu1.ponerAnimacionAlpha(true);


//*****************************************************************************
//Agregación de controles a la vista
//*****************************************************************************
  add(FondoMenu);
  add(graphContainer);
  add(areaTocable);
  add(botonOcultado_masV);
  add(botonOcultado_menosV);
  add(botonOcultado_masH);
  add(botonOcultado_menosH);
  add(ledActivador);
  add(texPanFrecCent);
  add(texMinNivel);
  add(texMaxNivel);
  add(texNivelDivision);
  add(texFrecDivision);
  add(texMinFrec);
  add(texMaxFrec);
  add(menu1);
```

Código 4.172: Agregación de distintos controles a la vista

Otro proceso de inicialización que ocurre en la vista es la realización del autoajuste inicial. Ello se consigue mediante el método de la vista *handleTickEvent* y un par de atributos de la vista: *Contadortick* e *Iniciado*. El código de este método es el siguiente:

```
void  FftView::handleTickEvent()
{
        if(!Iniciado)
        {
                Contadortick++;
                if(Contadortick%5==0)
                {
                        presenter->obtenerModelo()->ponerAutoAjuste();
                        Iniciado=true;
                }
        }
}
```

Código 4.173: Método handleTickEvent de la vista

Hay que recordar que el método *handleTickEvent* es un método que se ejecuta en cada actualización de pantalla. En el caso concreto de la vista, este método siempre está activo, no así en el caso de los controles que han de ser registrados en la instancia de la clase *Application* para poder recibir el evento *tick*. Este método *handleTickEvent* de la vista es ejecutado si ningún efecto durante cinco actualizaciones de pantalla, momento en el cual se ejecuta el autoajuste de pantalla y se indica que no se desea hacer ningún otro proceso temporizado desde la vista mediante la puesta a verdadero del atributo de la vista «*Iniciado*».

Otro método de gran importancia en la vista es el método actualizaSerieFFT:

```
1  void FftView::actualizaSerieFFT(unsigned char *registroFft, float minimo, float maximo,
2                                                              unsigned char canal)
3  {
4    Graph::Serie *serie;
5    serie=grafico.obtenSerie(0);
6    if(presenter->obtenerModelo()->obtenerTemporal())
7     {
8      serie->ponDatos(registroFft,2048);
9      serie->ponEscalaHor(serie->obtenEscalaHor());
10    }
11   else
12     {
13      serie->ponDatos(registroFft,1024);
14      serie->ponEscalaHor(serie->obtenEscalaHor());
15    }
16    grafico.invalidate();
17  }
18
```

Código 4.174: Método actualizaSerieFFT de la vista

La orden de actualizar los datos en el gráfico es originada en la aplicación final y llega al modelo a través de las colas del sistema; de allí al presentador a través del objeto *modelListener*. El presentador, entonces, ejecuta este método de la vista para que se produzca tal actualización. Dentro del método se accede a la serie a través de un puntero —línea 4—. Dependiendo si la representación es de los datos temporales o en el dominio de la frecuencia, se ejecutará el bloque entre las líneas 6 a la 10, para el primer caso, o el que hay entre las líneas 11 a la 15, para el segundo. Si la representación es de los datos temporales, se mostrarán en la gráfica todos los datos del registro —2048—, pero si lo que se representan son los datos de frecuencia, solo se mostrará la mitad —1024— ya que el espectro obtenido desde la señal temporal real es simétrico respecto del cero.

4.6.8 Objetos retrollamada —callback— y manipuladores de evento de la vista.

El mecanismo que posibilita la interacción en el sentido de usuario a entorno gráfico, es la clase retrollamada —*callback*—, mostrada en la sección 1.11.2. Este proyecto implementa una serie de objetos de esta clase que cumplen el siguiente diagrama genérico de secuencia:

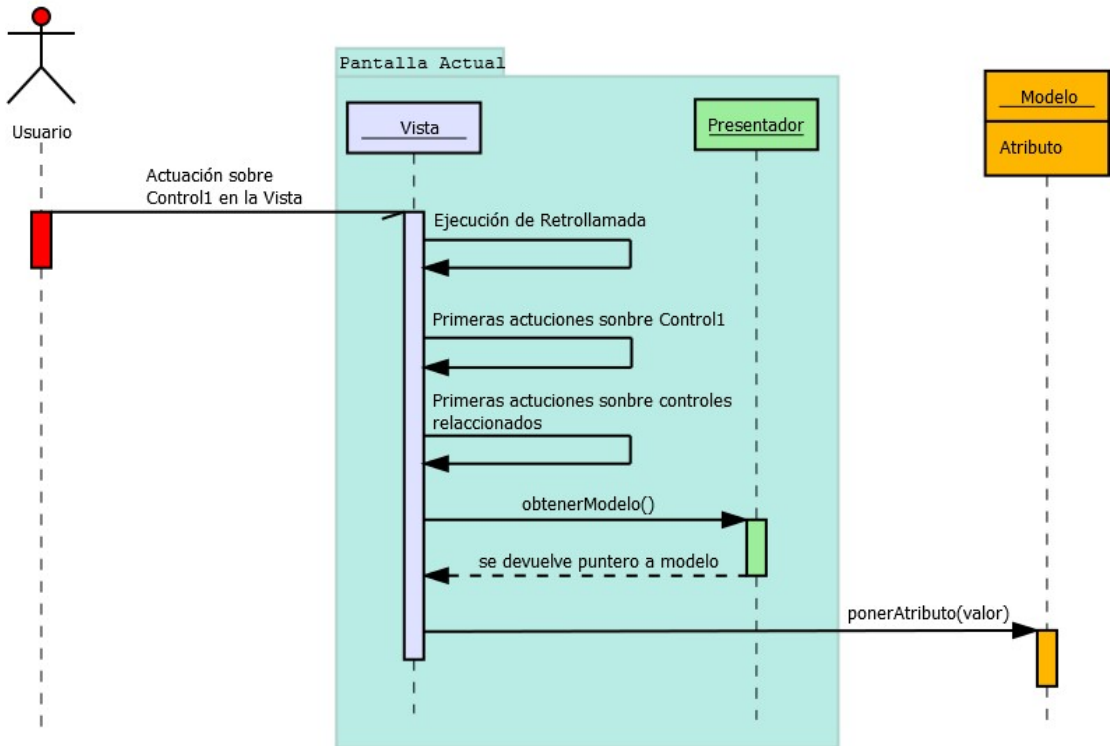


Ilustración 4.68: Diagrama de secuencia en la Vista

Al manipular un control en la vista por parte del usuario, se dispara la ejecución del objeto de tipo *callback* —retrollamada— que el control tiene asociado. Esta ejecución consiste en la llamada a un método de la vista que actúe como manipulador de evento, dando respuesta a la petición del usuario. Ya que, tanto el control tratado por el usuario, como el método que actúa de manipulador residen en la vista, el mecanismo puede ser visto como un mensaje recursivo —la vista genera y recibe el evento—. En la manipulación de este evento se realizan las primeras actuaciones en la gestión del mismo, como pueda ser la modificación de cierta propiedad en el control que lo dispara y las modificaciones en otros controles asociados. Sin embargo, la modificación de otros controles rara vez se hace en la función de manipulación del evento. Esta tarea se deja en manos del presentador ya que de esta forma se centralizan todas las modificaciones de controles relacionados. Cuando se ejecuta el manipulador de evento de la vista, por tanto, lo primero que se realiza es la modificación del propio control —este paso es opcional y no se realiza en todas las ocasiones— para a continuación solicitar la dirección del modelo al presentador, y de esta forma actualizar en el modelo el atributo correspondiente. Lo que ocurre a partir de aquí ya ha sido comentado en secciones anteriores; el modelo informa a la aplicación final y al presentador del

atributo que se acaba de modificar, el presentador actualiza los controles relacionados —ver sección 4.6.5— y la aplicación final actúa en consecuencia —dependiendo del atributo actualizado hará una u otra cosa, ver sección 4.5.9—.

Los objetos de tipo *callback* son declarados en la parte privada de la vista, existiendo tantos como controles de diferente tipo hay en la vista. Así, en este proyecto, los objetos de tipo *callback* declarados son los siguientes:

```
class FftView : public View<FftPresenter>
{
 public:
   . . .
 private:
  //*********************************************************************************
  //Declaración de eventos de usuario (Objetos retrollamada)
  //*********************************************************************************
  Callback<FftView, const AbstractButton &> BotonCallback;
  Callback<FftView, const AbstractButton &> BotonRadioSelecciondoCallback;
  Callback<FftView, const BotonEstados &> BotonEstadosCallback;
  Callback<FftView, const AbstractButton &, uint8_t> BotonOcultadoAnimacionEnCursoCallback;
  Callback<FftView, const SliderBar &, int16_t>DeslizadorCambioValorCallback;
  Callback<FftView, const ElementoMenu&, bool> MenuAnimacionTerminadaCallback;
  Callback<FftView, const JogWheel &, int16_t > RuedaAcabarCallback;
  Callback<FftView, const JogWheel &, int16_t > RuedaGiroCallback;
  Callback<FftView, const Cursor &, Cursor::CoordenadasEmpaquetadas >\
                                                    CursorCambioValorCallback;
  Callback<FftView, const Marcador&, const ClickEvent&> MarcadorCallback;
  Callback<FftView, const Marcador&> MarcadorActuandoCallback;
  Callback<FftView, const AbstractButton &, const DragEvent&> TouchAreaArrastradaCallback;
};
```

Código 4.175: Declaración de objetos callback en la vista

Se trata de plantillas donde se ha de determinar la clase concreta de la vista que los contiene, así como otros parámetros, que generalmente es la clase de referencia del objeto que puede disparar el evento. También puede tener otros parámetros que sirvan para que en la ejecución del evento se facilite más información. En el caso concreto de este proyecto están los objetos de retrollamada *BotonCallback*, para la manipulación de controles de tipo *AbstractButton* y descendientes, cuando son presionados; *BotonRadioSeleccionadoCallback*, para botones de opción cuando son seleccionados; *BotonesEstadosCallback*, para botones de estado cuando es modificado su estado; *BotonesOcultadoAnimacionEnCusroCallback*, para los botones de ocultación mientras están apareciendo o desapareciendo; *DeslizadorCambioValorCallback*, para los controles de deslizamiento cuando se modifica su valor; *MenuAnimacionTerminadaCallback*, para el menú cuando la animación del mismo ha concluido; *RuedaAcabarCallback*; para los controles de tipo rueda cuando se acaba de girarlos; *RuedaGiroCallback*, para los controles de tipo rueda mientras se está girando, *CursorCambioValorCallback*, para los cursores cuando al modificar su posición horizontal, cambia su posición vertical; *MarcadorCallback*, para los controles de tipo marcador cuando son presionados; *MarcadorActuandoCallback*, para controles de tipo marcador mientras está parpadeando y *TouchAreaArrastradaCallback*, para el área invisible que está encima del gráfico cuando se produce el arrastre sobre la misma.

Para poder enlazar cada objeto de retrollamada con su correspondiente método manipulador de evento de la vista, se necesita especificar, en su constructor, la dirección del objeto de tipo vista que lo contiene, así como la propia dirección del método que actuará de manipulador:

```
class FftView : public View<FftPresenter>
{
 public:
 FftView():
  . . .,
  BotonCallback(this, &FftView::Al_Boton),
  BotonRadioSeleccionDoCallback(this, &FftView::Al_BotonRadioSeleccionDo),
  BotonEstadosCallback(this, &FftView::Al_BotonEstados),
  BotonOcultadoAnimacionEnCursoCallback(this, &FftView::Al_BotonOcultadoAnimacionEnCurso),
  DeslizadorCambioValorCallback(this, &FftView::Al_DeslizadorCambioValor),
  MenuAnimacionTerminadaCallback(this, &FftView::Al_MenuAnimacionTerminada),
  RuedaAcabarCallback(this, &FftView::Al_RuedaAcabar),
  RuedaGiroCallback(this, &FftView::Al_RuedaGiro),
  CursorCambioValorCallback(this, &FftView::Al_CursorCambioValor),
  MarcadorCallback(this, &FftView::Al_Marcador),
  MarcadorActuandoCallback(this, &FftView::Al_MarcadorActuando),
  TouchAreaArrastradaCallback(this, &FftView::Al_TouchAreaArrastrada)

  { }
 . . .
 private:
 . . .
};
```

Código 4.176: Inicialización de los objetos callback en el constructor de la vista

Esta inicialización de los objetos *callback* se realiza en la lista de inicialización del constructor de la vista. Los métodos que actuarán como manipuladores de evento han de estar declarados en la parte pública de la vista, admitiendo como argumentos los mismos que los que posee su objeto *callback* correspondiente:

```
class FftView : public View<FftPresenter>
{
 public:
 FftView():
  . . .
  { }
   /**
     * Declaración de manipuladores de Eventos de los Widgets(componentes) de la vista
     * (FftView)
     */
  void Al_Boton(const AbstractButton &button);
  void Al_BotonRadioSeleccionDo(const AbstractButton& radioButton);
  void Al_BotonEstados(const BotonEstados &boton);
  void Al_BotonOcultadoAnimacionEnCurso(const AbstractButton &boton, uint8_t estado);
  void Al_DeslizadorCambioValor(const SliderBar &deslizador,int16_t valor);
  void Al_MenuAnimacionTerminada(const ElementoMenu &menu, bool estado);
  void Al_RuedaAcabar(const JogWheel &rueda, int16_t valor);
  void Al_RuedaGiro(const JogWheel &rueda, int16_t valor);
  void Al_CursorCambioValor(const Cursor &cursor,
                                          Cursor::CoordenadasEmpaquetadas coordenadas);
  void Al_Marcador(const Marcador& marcador, const ClickEvent& evento);
  void Al_MarcadorActuando(const Marcador& marcador);
  void Al_TouchAreaArrastrada(const AbstractButton &area, const DragEvent &evento);


 . . .
 private:
 . . .
```

Código 4.177: Declaración de manipuladores de evento para cada objeto callback

Como ejemplo de manipuladores de evento se muestra la siguiente porción del manipulador que gestiona todos los controles de tipo *Button*:

```
1  void FftView::Al_Boton(const AbstractButton &button)
2  {
3   . . .
4   if(&button == &botonActivacionLogaritmo)
5    {
6      bool estado=botonActivacionLogaritmo.getSelected();
7      presenter->obtenerModelo()->ponerEscalaLogaritmica(estado);
8    }
9
10  if(&button == &botonActivacionAdquisicion)
11   {
12    presenter->obtenerModelo()->ponerAdquiere(botonActivacionAdquisicion.getSelected());
13   }
14  . . .
15  if(&button == &botAutoajuste)
16   {
17    presenter->obtenerModelo()->ponerAutoAjuste();
18   }
19  . . .
20 }
```

Código 4.178: Porción del manipulador de evento para controles de tipo Button o descendientes

La estructura de cada manipulador es similar a la mostrada en este ejemplo. Consiste en bloques condicionales donde se comprueba el objeto causante del disparo del evento. Dentro de estos bloques se ejecuta el código correspondiente al control. Así para el caso de accionar el botón dedicado a cambiar la representación de los datos de logarítmica a lineal y viceversa —líneas de la 4 a la 8—, el estado de la selección del botón es tomada como estado para actualizar el atributo *EscalaLogaritmica* del modelo, lo que desencadena todo el proceso de comunicación dentro del entorno gráfico y entre este y la aplicación final, posibilitando todo el mecanismo del uso del entorno gráfico de usuario. Dentro de los bloques de condición se pueden comprobar otra serie de requerimientos, como por ejemplo, comprobar si algún parámetro enviado por el objeto *callback* al manipulador cumple cierta condición —no mostrado en esta porción de código—.

4.6.9 Código de utilidad en la vista

Al igual que sucedía con el modelo y el presentador, la vista dispone de código accesorio que le ayuda a hacer su labor. Este código está constituido por el atributo «*cadAux1*», un *array* 40 caracteres, y por los métodos «*cambiaPuntoPorComa*», «*actualizaTextoCursores*», «*obtenLuminancia*» y « *obtenColor*».

El atributo *cadAux1*, es utilizado para convertir información numérica en texto y dar formato a las cadenas en los controles de tipo texto, utilizando para ello, la función *snprintf* de la librería estándar:

```
1  snprintf(cadAux1, 15,"%.2f kHz",dx);
2  cambiaPuntoPorComa(cadAux1);
3  Unicode::strncpy(cadenaDeltaX, (char*)cadAux1, strlen((char*)cadAux1)+1);
4  texDeltaX.resizeToCurrentText();
```

Código 4.179: Utilización del atributo cadAux1 de la vista

Primeramente se convierte la información numérica —*dx*, en línea 1— en texto formateado mediante la función de la librería estándar *snprintf*. Luego, se realizan operaciones intermedias sobre *cadAux1*, como en este caso, convertir el separador decimal del punto a la coma —línea 2—. A continuación, esta cadena de texto simple —tipo *char*— es copiada en la cadena interna que el control de texto utiliza —cadena *cadenaDeltaX*, en línea 3—, y que es del tipo *Unicode*, una clase de cadenas con codificación Unicode que *TouchGFX* facilita para las cadenas de texto de los controles utilizados. Este tipo de cadenas dispone de varios métodos, como *strncpy* —mostrado en línea 3—, para copiar cadenas de texto de formato Unicode o *char*, sin embargo, en la versión 4.2 de TouchGFX, esta clase no dispone de un método para dar formato a una cadena desde una variable numérica —tipo *snprintf*—, algo que se solventa en versiones posteriores —la actual a la finalización de este trabajo, es la 4.8—, pero cómo se inicia este proyecto desde la versión 4.2, y se persigue compatibilidad con la misma, se mantiene la utilización de las cadenas de texto de los controles de esta manera.

Como ha sido mostrado en el código anterior, se utiliza el método de la vista *cambiaPuntoPorComa*, un método que se crea para poder expresar las cantidades numéricas en notación decimal en lengua Castellana. El código de este método es el siguiente:

```
1  void cambiaPuntoPorComa(char *cadena)
2   {
3     int i=0;
4     while(cadena[i])
5      {
6       if(cadena[i]=='.' && (cadena[i+1]>='0' && cadena[i+1]<='9'))
7          cadena[i]=',';
8       i++;
9      }
10  }
```

Código 4.180: Método cambiaPuntoPorComa de la vista

Recibe como argumento la cadena a modificar. Dentro del método, busca en la cadena el carácter «punto», pero teniendo la precaución de asegurarse que es el punto de separación decimal. Para ello comprueba que el siguiente carácter al punto represente un número. Si es así, sustituye el punto por la coma.

Para poder actualizar las lecturas de los cursores desde distintas partes del código, se crea la función *actualizaTextoCursores*:

```
1  void FftView::actualizaTextoCursores(void)
2   {
3    texXCursor1.resizeToCurrentText();
4    texXCursor2.resizeToCurrentText();
5    texYCursor1.resizeToCurrentText();
6    texYCursor2.resizeToCurrentText();
7    texDeltaX.resizeToCurrentText();
8    texDeltaY.resizeToCurrentText();
9
10   texXCursor1.invalidate();
11   texXCursor2.invalidate();
12   texYCursor1.invalidate();
13   texYCursor2.invalidate();
14   texDeltaX.invalidate();
15   texDeltaY.invalidate();
16  }
```

Código 4.181: Método actualizaTextoCursores de la vista

Existen, en total, seis controles de texto que hacen referencia a las lecturas de los cursores —todos ellos en el primer contenedor del menú «Cursores»—. Cada cursor tiene dos lecturas, una referente a la frecuencia, la horizontal —*texXCursorN*—, y otra al nivel, la vertical — *texYCursorN* —. Existen, además otros dos controles de

texto que muestran las diferencias entre las lecturas de cada cursor —*texDeltaX* y *texDeltaY*—. Este método se limita a redimensionar y refrescar estos controles de texto y debe ser llamado cuando hay una actualización en la posición de los cursores. Así, es llamado desde los manipuladores de evento *Al_MenuAnimacionTerminada*, que se ejecuta cuando la animación del menú ha finalizado, tras la selección de un elemento de menú; *Al_RuedaAcabar*, cuando se acaba de girar los controles que mueven los cursores; *Al_RuedaGiro*, mientras se giran estos controles; *Al_Marcador*, cuando el marcador que indica el estado de adquisición, indica que la adquisición está detenida, esto posibilita que al desplazar horizontalmente el gráfico se sigan actualizando las lecturas; y *Al_TouchAreaArrastrada*, que reitera la actualización que se realiza en el manipulador de evento anterior.

Para obtener el nivel de luminancia de un color dado, se crea el siguiente método:

```
1  uint8_t FftView::obtenLuminancia(uint16_t color)
2  {
3    uint8_t R, G, B;
4    float Rf, Gf, Bf, ct1, ct2, Lf;
5
6    R=Color::getBlueColor(color);
7    G=Color::getGreenColor(color);
8    B=Color::getRedColor(color);
9
10   if(R==0)R=1;
11   if(G==0)G=1;
12   if(B==0)B=1;
13
14   Rf=R/255.0f;
15   Gf=G/255.0f;
16   Bf=B/255.0f;
17
18   ct1=Rf/Gf;
19   ct2=Bf/Gf;
20
21   Lf=(Gf/3.0f)*(ct1+ct2+1);
22
23   return (uint8_t)((Lf*255.0f));
24 }
25
```

Código 4.182: Método obtenLuminancia de la vista

Para ello, se recibe el dato del color, en formato RGB565, al que calcular su luminosidad, y recupera sus componentes rojo, verde y azul en formato de ocho bits por color —a través de la clase de *TouchGFX Color*—. Luego, cada componente es pasada a formado de número flotante en el rango de cero a uno, para finalmente, a través de las componentes de color y sus relaciones, calcular la luminancia que es devuelta en el rango del uint8_t —de 0 a 255—. El hecho de la utilización de relaciones de colores —rojo respecto a verde y azul respecto a verde— es debido a que inicialmente se pensó en esta función para modificar la luminosidad de un color dado sin modificar su matiz y no utilizar expresiones más complejas como el formato HSV o HSL. Esta funcionalidad se delega en otra función, el método de la vista *obtenColor*:

```
1  uint16_t FftView::obtenColor(uint16_t color, uint8_t luminancia)
2  {
3    uint8_t R, G, B;
4    float Rf, Gf, Bf, ct1, ct2, Lf;
5
6    B=Color::getBlueColor(color);
7    G=Color::getGreenColor(color);
8    R=Color::getRedColor(color);
9
10   if(R==0)R=1;
11   if(G==0)G=1;
12   if(B==0)B=1;
13
14   Rf=R/255.0f;
15   Gf=G/255.0f;
16   Bf=B/255.0f;
17
18   ct1=Rf/Gf;
19   ct2=Bf/Gf;
20
21   Lf=luminancia/255.0f;
22   Gf=3*Lf/(ct1+ct2+1);
23   Rf=ct1*Gf;
24   Bf=ct2*Gf;
25
26   G = (Gf<=1)?(uint8_t)(Gf*255.0f):255;
27   R = (Rf<=1)?(uint8_t)(Rf*255.0f):255;
28   B = (Bf<=1)?(uint8_t)(Bf*255.0f):255;
29
30   return(Color::getColorFrom24BitRGB(R,G,B));
31 }
32
```

Código 4.183: Método obtenColor de la vista

Esta función tiene como argumentos el color a modificar su luminosidad y la luminosidad deseada, devolviendo el mismo matiz de color con la luminosidad solicitada. Para ello, tanto las componentes del color pasado, como la de la luminosidad deseada, son convertidas a números de coma flotante en el rango de cero a uno. Luego, se calculan las relaciones entre el rojo-verde y azul-verde. Estas relaciones han de mantenerse, a pesar de cambiar la luminosidad del color, si se quiere mantener su matiz. Con la luminosidad deseada y la expresión de la luminosidad se calcula la nueva componente verde, y a partir de ella, las componentes roja y azul. Luego, se restituyen los rangos de cada componente al del tipo uint8_t para finalmente ser devuelto el color modificado en formato RGB565 mediante el uso del método estático *getColorFrom24BitRGB* de la clase *Color* de *TouchGFX*.

Estos dos métodos son utilizados en aquellos lugares donde se usan controles de tipo barra de deslizamientos para seleccionar un color —método *obtenColor*— o un tono de gris —método *obtenLuminancia*—. Concretamente, ambos se utilizan dentro del menú *Pantalla*, en su segundo contenedor dedicado a la rejilla y al fondo del gráfico, así como en el menú *Cursores*, en el segundo y tercer contenedor, dedicados respectivamente a la apariencia del cursor1 y del cursor2. También son utilizados en los eventos de cambio de valor de los controles deslizadores utilizados en estos menús.

Presupuesto

Para el cálculo del presupuesto, se han tenido en cuenta los costes del equipamiento utilizado, tanto hardware como software, así como los derivados de las horas de trabajo en el desarrollo del hardware y software del producto final. También se han contemplado, como gastos de desarrollo, aquellos producidos por la elaboración de la documentación necesaria —como por ejemplo, esta memoria—. Todos los precios mostrados están indicados con el impuesto i.v.a añadido en vigor en el momento de realizar este documento. A continuación se detallan en tablas, los costes procedentes del equipamiento, los procedentes de la mano de obra, y finalmente el coste total.

Coste de Equipamiento

- **Hardware utilizado**

Concepto	Cantidad	Coste unitario (€/unidad)	Subtotal
Placa de desarrollo STM32F429I-DISCO	1	30 €	30 €
Display InnoLux AT05TN33	1	20 €	20 €
Gastos de aduana del display	-	34,16	34,16
Placa perforboard Eurocard 100mm x 160mm	1	60 €	60 €
Accesorios (conectores, cables y estaño)	-	40 €	40 €
PC portátil Lenovo con Windows 10	1	1100 €	1100 €
Coste total del hardware			**1284,16 €**

Tabla 24: Costes del Hardware utilizado

- **Software utilizado**

Concepto	Cantidad	Coste unitario (€/unidad)	Subtotal
Licencia Keil uvision 5.20 profesional para cortex M	1	10000 €	10000 €
Licencia de la librería gráfica TouchGFX 4.8 para 50000 dispositivos al año	1	15000 €	15000 €
Octave 4.2.0	1	0 €	0 €
Coste total del software			**25000 €**

Tabla 25: Costes del Software utilizado

Coste de mano de obra

Concepto	Cantidad (horas)	Coste unitario (€/hora)	Subtotal
Desarrollo Hardware	60	75 €	4500 €
Desarrollo Software	500	75 €	37500 €
Elaboración de Documentación	200	20 €	4000 €
Coste total de mano de obra			**46000 €**

Tabla 26: Costes de la mano de obra

Coste total

Concepto	Subtotal
Coste total del hardware	**1284,16 €**
Coste total del software	**25000 €**
Coste total de mano de obra	**46000 €**
TOTAL	**72284,16 €**

Tabla 27: Coste total del proyecto

Conclusiones y líneas futuras

La librería *TouchGFX* se presenta como una herramienta muy potente a la hora de hacer entornos gráficos de usuario. No obstante, hasta la última versión —4.8— no se disponía de un una utilidad gráfica donde poder diseñar las pantallas y los controles contenidos —algo que sí poseían competidores como *emWin* o *TARA*—. El desarrollo de este proyecto se inició en la versión 4.2 y se ha finalizado antes de estar este diseñador disponible. Debido a esto, el mayor tiempo empleado en el desarrollo ha sido debido a la codificación del entorno gráfico, maquetando la creación y posición de los controles del entorno directamente desde el código. A pesar de esto, el uso de una aplicación que suministre el código referente a la creación y posicionamiento, no elimina la tarea de dar el comportamiento deseado a cada control, es decir, sigue siendo necesario desarrollar los manipuladores de evento asociados. Es por ello que en futuros desarrollos de *TouchGFX* —con el diseñador de pantallas disponible—, y en general, en desarrollos con librerías gráficas comparables, se ha de valorar si la utilización de un entorno gráfico, para un sistema empotrado concreto, es una opción factible. En los criterios a valorar entrarán aspectos económicos, de tiempo de desarrollo y de lo que el usuario final espera de ellos.

TouchGFX es una librería diseñada en C++. Esto es una ventaja, ya que provee mucha flexibilidad. De hecho, se han practicado modificaciones en su implementación del patrón MVC debido a esta característica. Pero esto también es un inconveniente, ya que obliga a los desarrolladores de sistemas empotrados, acostumbrados a programar en lenguajes estructurados —tipo C— a aprender la programación orientada a objetos. Esto, unido a la obligación de conocimiento de otras librerías, como la propia del fabricante del micro, o de un sistema operativo concreto, hace la curva de aprendizaje muy pronunciada.

Por otro lado, en el desarrollo concreto realizado —aplicación que muestra la FFT sobre datos temporales capturados—, se utiliza como control para presentar la gráfica, uno de desarrollo propio. Este componente, aunque totalmente funcional y válido, presenta deficiencias a la hora de tratar la compresión gráfica —varios puntos ocupando la misma posición horizontal—. Ciertamente trata tal situación, pero no lo hace con técnicas como anti-aliasing o precisión subpixel, que sí son utilizadas por controles de tipo *Canvas* —controles gráficos especiales en *TouchGFX* para la representación vectorial—. Después del momento de comienzo de este proyecto, *TouchGFX* facilitó un control de representación de gráficas, tipo *Canvas*, en el repositorio de su página web. En futuros desarrollos se puede probar este nuevo componente, prestando especial cuidado a la velocidad de representación —ya que se trata de imágenes vectoriales que requieren mayor potencia de cálculo—. La adquisición de los datos temporales se ha realizado con una configuración conjunta de los tres ADCs que dispone el micro, con el objeto de obtener la máxima rapidez posible. Esto tiene el inconveniente que no permite disciplinar la adquisición con una señal de reloj, procedente de un temporizador, y con ello la imposibilidad de variar, de forma arbitraría, el periodo de adquisición —en el proyecto solo es posible elegir cuatro periodos concretos—. Para solventar esto, es posible la utilización de un único ADC, con la consiguiente pérdida de velocidad. Dependiendo de desarrollos futuros, es posible que esta pérdida de velocidad no sea un inconveniente. Este es el caso de una aplicación de tipo SDR —Software Defined Radio—.

El proyecto también representa los datos en el dominio del tiempo, de forma similar a un osciloscopio. Sin embargo, ya que su pretensión ha sido meramente de comprobación de la forma de onda, sin querer desarrollar un osciloscopio, carece de disparo y escalado de los datos. En desarrollos futuros es posible incorporar estas características. Por ejemplo, en el caso del disparo, podría implementarse con los dos niveles de umbral de captura que dispone el periférico ADC del micro STM32F429.

Bibliografía y referencias

[1] TouchGFX User's Manual, Draupner Graphics A/S. 2017.

[2] Página web de TouchGFX: www.touchgfx.com, www.draupnergraphics.com.

[3] Código fuente disponible en la librería TouchGFX 4.2 —archivos de cabecera—.

[4] ST RM0090, Manual de referencia STM32F405/415, STM32F4 7/417, STM32F427/437, STM32F429/439 avanced ARM –based 32-bit MCUs.

[5] ST Datasheet production data STM32F427xx/STM32F429xx ARM Cortex-M4 32 MCU+FPU.

[6] ST UM1670 User manual: Discovery kit with STM32F429ZI MCU.

[7] ST UM2052 User manual: Getting started with STM32 MCU Discovery Kits software development tools.

[8] Página web de STMicroelectronics: www.st.com.

[9] Cortex-M4 Device Generic User Guide ARM DUI 0553A (ID12610) 2010.

[10] Página web FreeRTOS: www.freertos.org.

[11] ARM IHI 0042F: Procedure Call Standar for ARM Architecture. Release 2.10. 2015.

[12] The Scientist and Engineer's Guide to Digital Signal Processing. California Technical Publishing, Second Edition. Smith, S. W., 1999. ISBN: 0-9660176-6-8.

[13] On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform. Fredric J. Harris. IEEE, vol 66, pp. 51-83, Jan. 1978.

[14] Nota aplicación Tecktronix 55W-8815-2 :FFT Applications for TDS Oscilloscopes, 05/05.

[15] Procesamiento Digital de Señales, Transforma de Fourier y el algoritmo FFT, — FCEFyN— UNC, Universidad Nacional de Córdoba, Argentina.

[16] National Instruments Application Note 041: The Fundamentals of FFT-Based Signal Analysis and Measurement. Michael Cerna and Audrey F. Harvey. Julio 2000.

[17] IME Report 01/2009: How to use the FFT for signal and noise simulations and measurements. Institute of Microelectronics, University of Applied Sciences Northwestern Switzerland, School of Engineering. Hanspeter Schmid.

[18] Freescale Application Note AN4255: FFT-Based Algoritm for Metering Applications. Ludek Slosarcik. Rev. 4, 2015.

[19] Discrete-Time Signal Processing. Second edition. Alan V. Oppenheim, Ronald W. Schafer, John R.Buck. ISBN 0-13-754920-2, 1998.

[20] Documentación de ayuda y código fuente de la librería CMSIS 5.1.0.

[21] Programación en Borland C++ 3.X, ISBN: 84-7614-472-5, Lee Atkinson, Mark Atkinson, ed. Anaya, 1993.

[22] Desarrollo de Vídeojuegos. Un Enfoque Práctico. Julio 2014, ISBN: 978-84-942382-9-1. Escuela Superior de Informática, Universidad de Castilla-La Mancha.

[23] Página web de Easing Function: http://easings.net.

[24] A050-33-TT-01 LCD Module Specification Innolux AT050TN33, Version 01, 2008.

[25] A043-24-TT-11 LCD Module Specification Innolux AT043TN24 V.1, Version 01, 2008.

[26] SAEF Technology Specification of LCD Module No.: DataSheet SF-TC240T-9370A-T, REV A, 2012.

[27] ILITEK ILI9341 a-Si TFT LCD Single Chip Driver 240RGBx320 Resolution and 262K color Specification, Version: V1.10.

[28] Datasheet STMPE811, 8-bit port expander with advanced touchscreen controller, June 2008.

[29] Cypress MicroSystems Application Note AN2173: Touch Screen Control and Calibration — Four Wire, Resistive. Version: 4.1. Svyatoslav Pali. 2004.

[30] Nota de aplicación Microchip AN1368: Developing Embedded Graphics Applications using PIC Microcontrollers with Integrated Graphics Controller. Pradeep Budagutta. 2011.

[31] Microchip AR1000: Series Resistive Touch Screen Controller. 2009.

[32] Texas Instruments Application Report SLAA384A: 4-Wire and 8-Wire Resistive Touch-Screen Controller Using the MSP430. 2010.

[33] ST Application note AN4031: Using the STM32F2, STM32F4 and STM32F7 Series DMA controller. Rev 3. June 2016.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá