Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Diseño de Controlador MIDI con Comunicación Bluetooth en Microcontrolador MSP430

Autor: Eduardo Zafra Ratia
Tutor: Manuel Ángel Perales Esteve

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Diseño de Controlador MIDI con Comunicación Bluetooth en Microcontrolador MSP430

Autor:
Eduardo Zafra Ratia

Tutor:
Manuel Ángel Perales Esteve
Profesor Contratado Doctor

Departamento de Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Diseño de Controlador MIDI con Comunicación Bluetooth en Microcontrolador MSP430

Autor: Eduardo Zafra Ratia

Tutor: Manuel Ángel Perales Esteve

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

Agradecimientos

Agradecer en primer lugar, a mi familia, en especial a mis padres, por todo el esfuerzo y el apoyo que me han brindado, permitiendo la posibilidad de dedicarme al estudio durante estos años de carrera.

Así mismo, me gustaría agradecer a Manuel Perales todos los consejos e indicaciones que me han permitido llevar a cabo la realización de este proyecto.

Eduardo Zafra Ratia

Sevilla, 2016

Resumen

El protocolo MIDI surge a principios de la década de los ochenta ante la necesidad de desarrollar un estándar de audio digital que simplificara el trabajo con numerosos instrumentos electrónicos, estableciendo un canal de comunicación entre los mismos. Para ello, el protocolo se basa en el envío de una serie de mensajes de evento y diversos parámetros de control que permiten la generación de sonidos y el ajuste de estos parámetros. [1]

Pese al paso de los años, el estándar MIDI sigue plenamente vigente en la actualidad gracias a su sencillez y a la flexibilidad que otorga la posibilidad de interconectar instrumentos y controladores independientemente del fabricante. Estas características, así como el pequeño tamaño de los archivos MIDI en comparación a un archivo de audio muestreado, han permitido también, la expansión de este protocolo más allá de los estudios de grabación profesionales.

Por estos motivos, se justifica el desarrollo del presente proyecto mediante el cual se abordará el diseño de un controlador MIDI que permita controlar remotamente diversos parámetros de un instrumento o sintetizador MIDI. Nos basaremos en el sistema microcontrolador de Texas Instruments MSP430G2553 tanto para el emisor de los comandos como para el receptor. Para establecer la comunicación remota se harán uso de sendos módulos Bluetooth que permitan el paso de mensajes a distancia.

Para la interfaz de control, se hará uso de una pantalla táctil resistiva que se conecta al microcontrolador emisor y a través de la cual se podrán manipular los diversos comandos para los distintos canales disponibles.

Abstract

The MIDI protocol was created in the early eighties in order to develop a new digital audio standard that simplified the use of several electronical instruments at the same time, by establishing a communication channel between these devices. To accomplish this goal, the protocol is based in the transmission of event messages and control parameters that are capable of generating sounds and adjust de value of these parameters.

Despite the passing years, the MIDI standard is still vastly used due to its simplicity and its flexibility, allowing the interconnection of many instruments and controllers regardless of the manufacturer. These features, along with the small size of MIDI files, compared to a sampled audio file, are the cause to the great extension of MIDI devices beyond professional recording studios.

All these reasons justify the development of a wireless MIDI controller that allows the user to remotely modify certain parameters of a MIDI synthesizer. The project will be based in the Texas Instruments MSP430G2553 microcontrollers, for both the emitter and the receiver. To establish a wireless communication, two Bluetooth modules will be used.

The control interface will be implemented with a resistive touchscreen that is connected to the emmiter microcontroller and will allow the user to change the value of the different parameters for the available channels.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

bps	Bits por segundo
CC	Control Change
dBm.	Decibelio-milivatio
IoT	Internet of Things
Vcc	Tensión de alimentación
GND	Tierra
CLK	Señal de Reloj
V_{OH}	Mínimo voltaje de salida para nivel alto
MOSI	Línea de Master Out – Slave In
MISO	Línea de Master In – Slave Out
RXD	Recepción de datos
TXD	Transmisión de datos
ksps	Kilo Samples por segundo
<<	Desplazamiento de bits a la izquierda
>>	Desplazamiento de bits a la derecha
POSX	Posición en el eje X de la pantalla en píxeles
POSY	Posición en el eje Y de la pantalla en píxeles

1 INTRODUCCIÓN

Como se ha explicado brevemente en el resumen del proyecto, los dispositivos y aplicaciones musicales que hacen uso del estándar MIDI en la actualidad son numerosos. Basta revisar cualquier tienda especializada para encontrar una gran variedad de instrumentos, sintetizadores, secuenciadores y controladores MIDI. [2] [3]

También son destacables muchas otras aplicaciones basadas en MIDI que más allá de entornos meramente profesionales, han estado y están presentes en sectores más cotidianos de la electrónica de consumo, como pueden ser los politonos, que consiguieron una gran popularidad en la década pasada en los albores de la telefonía móvil, los videojuegos musicales, o aplicaciones tan conocidas como la GarageBand de Apple.

Existen, además, interfaces MIDI-USB que permiten la conexión de estos dispositivos con un PC, y utilizar diversas herramientas software de manera que un ordenador puede convertirse en un controlador o instrumento MIDI más, que puede comunicarse perfectamente con otros dispositivos MIDI con normalidad.

De esta manera, el objetivo de este proyecto será diseñar un controlador MIDI, mediante el cual se puedan enviar comandos y actuar sobre un instrumento MIDI para modificar distintos parámetros del sonido.

Usualmente, los controladores MIDI disponibles en el mercado son superficies de control que poseen una serie de botones, sliders, encoders y otros elementos asociados a parámetros de control MIDI, de manera que al accionar alguno de estos componentes, se envía el mensaje correspondiente para actuar sobre alguna característica del sonido en tiempo real. El nuevo valor del parámetro se mantendrá constante en el receptor sin necesidad de refrescarlo hasta que éste sea manipulado nuevamente.

Para este proyecto, sin embargo, se hará uso de una pantalla táctil resistiva que constituya la interfaz mediante la cual el usuario pueda interactuar con el controlador mediante la representación en la misma de los diferentes controles disponibles. Se hará uso de la pantalla VM800 del fabricante FTDI, la cual permite gracias a su coprocesador gráfico, poder alcanzar diseños bastante complejos sin necesidad de utilizar un microcontrolador de grandes prestaciones para actuar sobre ella. Esto permitirá, abordar el diseño de todo el proyecto con dos microcontroladores MSP430G2553 de Texas Instruments, que destacan por su bajo coste, su alta disponibilidad en el departamento de Ingeniería Electrónica, y una serie de prestaciones que serán detalladas y examinadas a fondo en capítulos posteriores, pero que en todo caso son suficientes para desarrollar el presente proyecto plenamente.

Será igualmente un punto clave del proyecto la necesidad de dotar al controlador de comunicación remota que permita el uso a distancia del controlador y la movilidad del mismo respecto del instrumento a controlar. Para ello, se hará uso de la tecnología Bluetooth, un estándar de comunicación que aparece a finales de siglo pasado como una solución ideal para comunicaciones a corta distancia, y que ha sido ampliamente mejorado en sus diferentes versiones a lo largo de los años. En sus versiones más actuales, como la 4.0, permite velocidades de hasta 36MB/s, e intruducen nuevas especificaciones de bajo consumo como el Bluetooth Low Energy (BLE). Estas nuevas versiones siguen siendo compatibles con las versiones más antiguas del protocolo, permitiendo que permanezcan en el mercado dispositivos Bluetooth diseñados para especificaciones anteriores, que no requieren mayores velocidades. Es el caso de los módulos Bluetooth HC-05 que se utilizarán. Estos siguen el estándar Bluetooth 2.0 + EDR (Enhanced Data Rate) que permite una velocidad de transferencia de hasta los 3MB/s y un alcande de hasta unos 10m. [4]

De este modo, los dos módulos bluetooth constituyen la conexión entre los dos grandes bloques a diseñar en este proyecto. Por un lado, el bloque receptor, que constará de un microcontrolador y su módulo HC-05 correspondiente, ambos conectados mediante una conexión serie UART. Este bloque simplemente se encarga de transmitir lo que le llega por el módulo Bluetooth a la entrada. En su salida, se conecta a una placa que toma la información transmitida y la adapta al estándar MIDI en su nivel físico, desembocando en un conector MIDI. Con un software adecuado y un adaptador MIDI-USB, se podrá comprobar fácilmente el funcionamiento adecuado del controlador, utilizando un PC como instrumento a controlar.

Por otro lado, el bloque emisor, que está compuesto de un microcontrolador, el módulo bluetooth correspondiente, y la pantalla táctil resistiva que implementará la interfaz, haciendo esta última, uso del módulo SPI del microcontrolador.

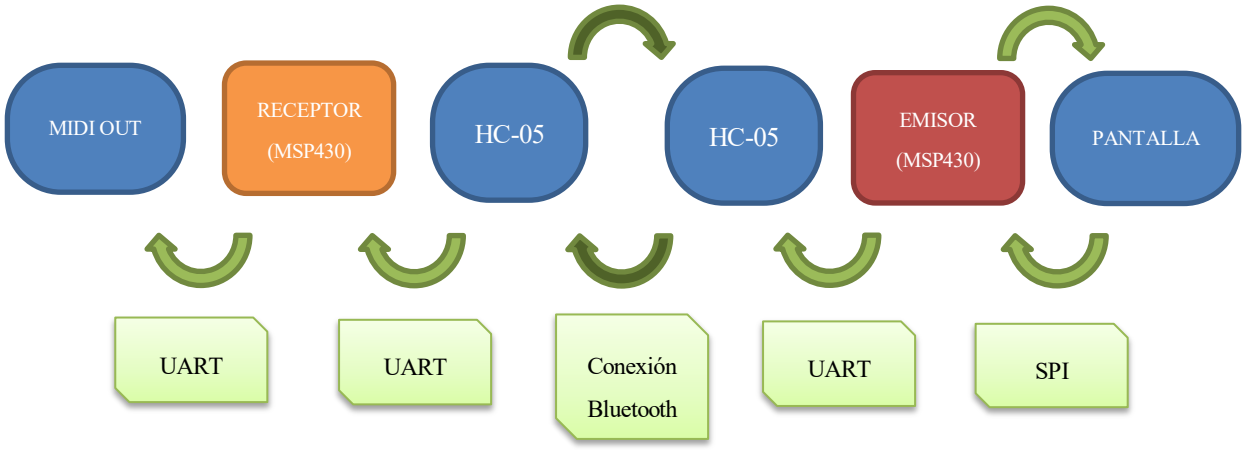

Figura 1-1. Esquema básico del proyecto.

2 ESTÁNDAR MIDI

A lo largo de este capítulo, se describirán las principales características del protocolo MIDI. Tanto desde el punto de vista del Software (tipos de mensajes, estructura de un mensaje… etc), como desde el punto de vista del Hardware (conectores, dispositivos…etc). También se hará un breve repaso de la vigencia y la presencia de esta tecnología en la actualidad.

2.1. Introducción

El estándar MIDI nace en 1983 ante la necesidad de desarrollar un protocolo que permitiera comunicar diversos instrumentos entre sí, dotando de una mayor flexibilidad a la hora de poder utilizer los instrumentos electrónicos de la época. Es producto del acuerdo y colaboración de los fabricantes de la época. [5]

La gran aportación de este protocolo a la industria, fue facilitar el desarrollo de grabaciones y ediciones complejas, reduciendo en muchos casos la inversión necesaria para llevar a cabo este tipo de proyectos, eliminando todas las complicaciones y costes extra que implicaba la ausencia de un estándar fuertemente implantado hasta ese momento. Con un simple sintetizador con teclado MIDI, se poseía ahora una versatilidad mucho mayor que en años anteriores, con la consiguiente disminución de costes en mano de obra (músicos) e instrumentos diversos. En definitiva, hizo que la creación de música fuera un mercado mucho más accesible y asequible.

Así mismo, la reproducción de música en directo también pudo disfrutar de numerosas ventajas. Se facilita el transporte de los instrumentos, el montaje de los mismos…etc, y se abre una nueva gama de posibilidades al músico que toca en directo.

De manera muy básica, una grabación MIDI consiste en una serie de instrucciones a las que les corresponden una serie de sonidos. Estas instrucciones dan información de la nota a tocar, el canal…etc. Existen, también una serie de mensajes que permiten variar ciertas propiedades del sonido, como pueden ser el volumen, el balance, el instrumento, o cualquier otro efecto que puede depender del dispositivo MIDI en cuestión.

2.2. Características Software

2.2.1 Transmisión de datos

Los mensajes MIDI viajan asíncronamente y en serie, sin señal de reloj, a una velocidad de 31 250 bps. Esta velocidad se debe simplemente a que es división exacta de 1MHz.

El protocolo se basa en una UART con 1 bit de start y otro de stop. No hay bit de paridad. De esta menera, la línea se mantiene a nivel alto en reposo, y el mensaje correspondiente comienza siempre con el bit de start (0 lógico). El dato correspondiente se transmite desde el bit LSB al MSB, finalizando la transmisión con el bit de stop (nivel alto). [6]

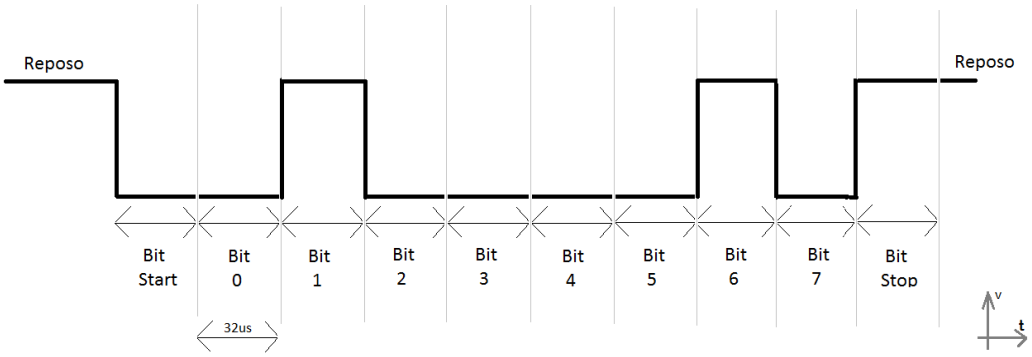

Figura 2-1. Ejemplo transmisión dato "01000010"

2.2.2 Estructura de un mensaje MIDI.

Un mensaje MIDI completo tiene asociados tres bytes completos. El primero es el byte de Estado (Status byte), y los otros dos son los bytes de datos (Data bytes), que complementan la instrucción del primero. En algunos casos es sólo necesario un byte de datos. [1]

En todo caso, los bytes de estado comienzan con un 1 lógico, mientras que los bytes de datos siempre comienzan con el primer bit a nivel bajo. De esta manera, tendríamos los 7 bits restantes del byte para definirlo claramente. Es decir, hasta 127 valores distintos.

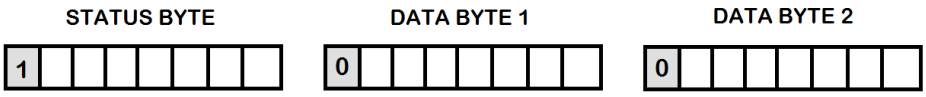

Figura 2-2. Ejemplo mensaje MIDI

Antes de profundizar en las estructuras de los distintos tipos de mensajes, es conveniente abordar el concepto de canal. En MIDI, existen 16 canales distintos que pueden recibir y transmitir las instrucciones correspondientes de manera separada.

Atendiendo a este concepto de canal, los mensajes MIDI pueden clasificarse de manera básica en dos tipos distintos: Los mensajes de canal y los mensajes de sistema. En función de esta clasificación, el byte de estado tendrá una forma distinta.

Los mensajes de canal, tienen como destinatario un canal en concreto. De este modo, si por ejemplo procedemos a un cambio del volumen del sonido, éste sólo se aplicará al canal correspondiente, quedando el volumen de los demás canales inalterado. Como existen 16 canales, se utilizan los 4 bits LSB para indicar el canal. De los 4 bits restantes, el más significativo siempre valdrá 1 para indicar que es un byte de estado, mientras que los otros 3 bits sirven para indicar el tipo del mensaje. Como tenemos 3 bits, se puede deducir que habrá 8 tipos de mensajes de canal distintos. Siendo el caso en el que los 3 bits valen “111”, un mensaje de sistema.

Para los mensajes de sistema, el destinatario son todos los canales. En este caso, como no es necesario indicar un canal específico, los 4 bits LSB se utilizan para indicar el mensaje. Los 4 bits MSB valen siempre “1111” (0xF). Para esta clase de mensajes existen, por lo tanto, 16 tipos diferentes.

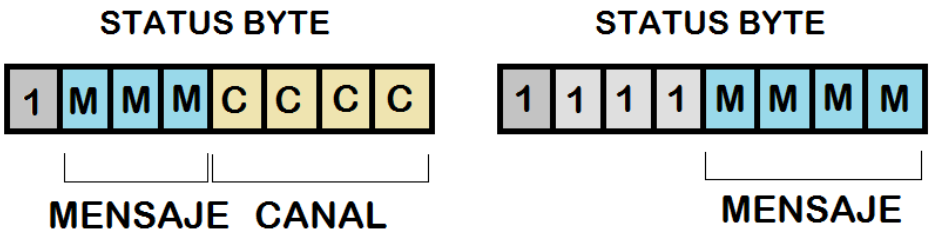

Figura 2-3. Mensajes de canal y de sistema

Teniendo en cuenta todo esto, los ocho tipos de mensajes de canal disponibles se pueden resumir en el siguiente cuadro:

Tipo mensaje	Status byte	St. Byte (hex)	Data byte 1	Data Byte 2
Note off	1000 nnnn	8n	Altura	Velocidad
Note on	1001 nnnn	9n	Altura	Velocidad
Poly. Aftertouch	1010 nnnn	An	Altura	Presión
Control Change	1011 nnnn	Bn	Nº de Control	Valor
Program Change	1100 nnnn	Cn	Nº de Programa	—
Chan. Aftertouch	1101 nnnn	Dn	Presión	—
Pitch Bend	1110 nnnn	En	LSB Byte	MSB Byte
System Message	1111 mmmm	FM		

Tabla 2–1. Tipos de mensajes en MIDI [1]

Los comandos “note off” y “note on”, sirven para desactivar o activar respectivamente la nota correspondiente al valor entre 0-127 que se proporciona en el Data byte 1. A mayor valor, estaríamos ante una nota más aguda. El data byte 2 corresponde a la velocidad con la que se pulsa o se deja de pulsar la tecla en cuestión. Ante valores altos, se podría apreciar un pequeño aumento del volumen y de la frecuencia (sonido más agudo), como pasaría en un instrumento real.

Para hacer un uso correcto de estos comandos, es necesario conocer el tipo de instrumento que se está tocando. En instrumentos monofónicos, un nuevo comando “note on” implica irremediablemente el final de la nota anterior sin necesidad de un “note off”. Para instrumentos polifónicos, esto no es así ya que es posible tocar varias notas a la vez. En todo caso una nota, acaba si se manda el “note off” correspondiente. Así mismo, dependiendo de la característico del instrumento, la nota se mantiene en el tiempo durante una duración determinada que depende del instrumento. Un instrumento polifónico puede tener un límite en el número de notas que se pueden tocar a la vez.

El mensaje “polyphonic aftertouch” sirve para indicar la presión que se realiza sobre una tecla determinada. “Channel aftertouch” es similar, pero da el valor de presión para todas las notas que se toquen en el canal indicado. Este último se encuentra de manera más frecuente en la mayoría de instrumentos que el primero.

Esto es debido a que teclados con "polyphonic aftertouch" necesitan de mecanismos más complejos y además generan una mayor cantidad de datos MIDI que puede ser excesiva en el caso de los instrumentos más antiguos.

El comando Pitch Bend sirve para variar la frecuencia fundamental de la nota que se está tocando. Esta variación se cuantifica con los dos Data Bytes, siendo el primero el correspondiente a los bits menos significativos del valor y el segundo a los más significativos, dando un total de 14 bits para codificar el valor de Pitch. De esta manera tendríamos valores comprendidos entre 0 y 16 384, siendo 8 192 el valor central, para el cual no habría variación del tono. Se puede optar por un control más grueso del Pitch, dejando los 7 bits LSB a 0 en todo momento y variando el valor de los 7 bits MSB.

Figura 2-4. Control habitual de rueda para Pitch Bend[1]

La instrucción Program Change sirve básicamente para cambiar el instrumento del canal correspondiente. Para ampliar el número de instrumentos que habría originalmente disponibles (0-127), estos se disponen en bancos de hasta 128 instrumentos cada uno, pudiendo saltar de un banco a otro mediante el Control Change correspondiente.

Por último, los comandos Control Change, son de gran importancia que son un conjunto de comandos que permiten actuar y modificar numerosos parámetros de control. Con el primer Data byte se selecciona el Control Change que se quiere variar (por ejemplo, Modulation), y mediante el segundo Data byte proporciona el valor del parámetro de control. Como es de esperar, existen hasta 127 Control Change distintos. Algunos de ellos están muy estandarizados en todos los dispositivos MIDI debido a su importancia y frecuencia de uso. Otros Control Change, quedan sin asignar y dependen del instrumento a utilizar. A continuación, se citan algunos de los controles más habituales, obviando aquellos menos estandarizados y cuya función puede variar enormemente de un dispositivo a otro:

- CC #0: Bank Select. Permite cambiar entre los diferentes bancos de instrumentos disponibles. En teoría, hasta 128 bancos disponibles. Estos dependerán del dispositivo.
- CC #1: Modulation. Control muy frecuente en teclados que suele implementarse con una rueda similar al Pitch Bend. Puede modificar diversos efectos del sonido en función del instrumento. Es habitual que esté asociado a vibrato.
- CC#2: Breath. Permite articular las distintas notas, dando un efecto parecido al de los instrumentos de viento.
- CC#7: Volume. Cambia el volumen del canal correspondiente. Es importante configurar este valor atendiendo también al valor de volumen del resto de canales.
- CC#8: Balance. Ajusta el balance del sonido entre el altavoz izquierdo y derecho. Permite jugar con la localización del instrumento ajustando la ganancia o atenuación correspondiente para la señal que llega a cada altavoz
- CC#10: Pan. Efecto similar al de Balance, pero sin jugar con ganacias a los distintos altavoces. Literalmente trata de trasladar una señal estéreo de un altavoz a otro. Con una señal mono, el comportamiento es el mismo que el de Balance.

[1] https://it.wikipedia.org/wiki/Rotella_di_modulazione a fecha de Agosto de 2016

- CC#11: Expression Controller: Al igual que el CC#7, juega con el volumen de la pista. Trabaja en conjunción con éste, de manera que se suele utilizar para hacer crescendos y decrescendos para una serie de notas en concreto, sin alterar el valor de volumen general.
- CC#64. Pedal Sustain. Indica la pulsación o no de un pedal típico en numerosos instrumentos. Da el efecto de que las notas se mantienen más en el tiempo.

Los controles del 120 al 127, son una serie de controles reservados llamados "Mensajes de Modo". Estos son:

- CC#120. All sound off. Apaga todos los sonidos del sistema lo más rápido posible.
- CC#121. Reset all controllers. Resetear todos los controles a su valor por defecto. Puede ser útil al acabar una interpretación en la que se han manipulado controles cuyos cambios no se quieren conservar para el siguiente tema.
- CC#122. Local control. Si se desactiva (value=0), se ignora cualquier pulsación en el teclado o los controles, y sólo se atenderán los mensajes que lleguen por la conexión MIDI.
- CC#123. All notes off. Desactiva todas las notas. Útil si no están llegando los "note off" adecuadamente. Los siguientes CC implican siempre un "All notes off".
- CC#124. Omni mode off. Sólo se responde a los datos que llegan por un canal particular.
- CC#125. Omni mode on. Se responde a lo que llega independientemente del canal
- CC#126. Poly mode off. Modo monofónico.
- CC#127. Poly mode on. Modo polifónico.

Finalmente, restarían los mensajes de sistema. Dentro de este grupo, podemos encontrar tres tipos distintos. Estos son, los mensajes de sistema comunes, los de tiempo real y los exclusivos. Los tres son recibidos por todos los canales.

2.3. Características Hardware

Desde el punto de vista físico, una conexión MIDI se implementa mediante conectores tipo DIN de 5 pines. El uso de estos pines depende del tipo de conexión que se realice. Se diferencian tres tipos de conexión:

- MIDI IN: Recepción de datos.
- MIDI OUT: Transmisión de datos.
- MIDI THRU: Transmite lo recibido por IN.

La conexión mas habitual es conectar dos dispositivos siendo uno MIDI OUT y otro MIDI IN, de manera que uno actuaría como "maestro", llevando a cabo la transmisión de datos, a través de su conector MIDI OUT y el otro actuaría como "esclavo", esperando la recepción de los mismos, a través de su conector MIDI IN, que se encuentra conectado al MIDI OUT del maestro. Los conectores MIDI THRU, permiten un control en cascada de numerosos dispositivos a partir de un solo controlador.

Figura 2-5. Conectores DIN de 5 pines MIDI

De estos 5 pines, no todos tienen una función determinada por el estándar. Los pines 1 y 3 (ver figura 2-6), no tienen función asignada. En el caso de que el conector sea un MIDI OUT, el pin 5 va a alimentación y el pin 2 a tierra, para acometer la función de blindaje. El pin 4 corresponde con la señal a transmitir. La distribución es la misma en el caso de MIDI THRU.

Para un conector MIDI IN, se utilizan sólo los pines 4 y 5 como entrada diferencial. Mediante un circuito optoacoplador se detecta la circulación de corriente, abriendo o cerrando el circuito a la salida según corresponda para la lectura del mensaje. [6]

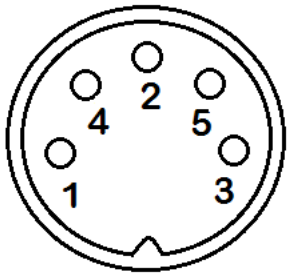

Figura 2-6. Esquema de pines conector MIDI

2.4. Estado del Arte

En la actualidad, y pese a la aparición de otros estándares competidores como el OSC (Open Sound Control), el estándar MIDI sigue teniendo un peso muy importante en la industria.

En el caso del estándar OSC (creado en 2002), se obtienen en principio numerosas ventajas. Sigue como esquema el protocolo IP de Internet para llevar a cabo la comunicación entre los distintos dispositivos. Esto, les dota de conectividad a la red, lo cual puede ser importante para aplicaciones del "Internet of Things". Como protocolo en la capa de transporte, utiliza el protocolo UDP (no orientado a conexión), más apropiado que el protocolo TCP para aplicaciones de tiempo real, donde la comprobación de errores puede perjudicar la temporalidad del sistema introduciendo mayor latencia y jitter. [7]

El propósito en su creación, era desarrollar un estándar mucho más rápido, flexible, accesible y de propósito más general que MIDI. Además, proporciona una gama de parámetros de control y una resolución de los mismos mucho más amplia (rango mucho más amplio que el 0-127 típico de MIDI).

Todo esto, no evita el hecho de que superar la implantación de MIDI en la industria es algo prácticamente inabordable, ya que la proliferación de toda clase de dispositivos MIDI desde la creación del estándar y el propio crecimiento de la industria, dificulta enormemente un acuerdo similar al de 1983 entre los distintos fabricantes para adoptar un nuevo estándar. Incluso aunque se salvara ese obstáculo, seguiría habiendo un grandísimo número de dispositivos MIDI que seguirían estando presentes, y seguirían siendo utilizados, lo que requeriría al final una conversión entre un estándar y otro, dificultando dejar atrás el estándar MIDI de manera completa. Este paso no se realizará hasta que verdaderamente no se desarrolle un nuevo estándar cuyas prestaciones y ventajas sean lo suficientemente atractivas como para dar ese gran salto necesario.

De momento, la realidad es que el protocolo MIDI sigue siendo la referencia dentro del mercado de instrumentos digitales, siendo posible encontrar en tiendas todo tipo de aparatos y aplicaciones de audio que siguen el estándar MIDI. Básicamente estos dispositivos se pueden clasificar en dos grandes grupos: secuenciadores y sintetizadores.

Un secuenciador, es cualquier dispositivo capaz de generar una secuencia de mensajes MIDI. Por ejemplo, tocar una nota, cambiar programa, modificar un control change… etc. Por el contrario, un sintetizador se encarga de generar los sonidos correspondientes a partir de la información generada en un secuenciador. Ambos se pueden encontrar como dispositivos hardware o como herramientas software. Es bastante frecuente que un mismo dispositivo o aplicación pueda llevar a cabo ambas funciones.

Más allá de esta clasificación, podemos encontrar dispositivos MIDI que simulan instrumentos reales, como baterías, violines, instrumentos de viento...etc. Algunos ejemplos pueden ser la gama de baterías electrónicas TD de Roland, los violines electrónicos SV-150 y superiores de Yamaha, o el instrumento de viento EWI4000S de Akai.

Figura 2-7. Instrumentos MIDI[2]

Lo más habitual es encontrar instrumentos en forma de teclados. Muchos de estos, incluyen por defecto pequeños controladores asociados a comandos MIDI para cambiar aspectos del sonido. A parte, se pueden encontrar superficies de control más exhaustivas que ofrecen un gran número de sliders, encoders u otros métodos de interacción para manipular diferentes parámetros. Los ejemplos en este caso son muy numerosos ya que son muchos los fabricantes que eligen el formato de teclado para hacer instrumentos y controladores MIDI. Se pueden citar como ejemplo: Behringer UMX250, Roland A-49, Akai MPK261, la serie S de teclados Komplete Kontrol de Native Instruments…etc. [2] [3] Se puede observar como todos estos aparatos incluyen una amplia gama de deslizadores, controladores rotatorios, palancas y botones que permiten manipular los controles MIDI a los que están asociados. En muchas ocasiones, permitiendo la programación por parte del usuario de estos elementos para asociarlos a parámetros MIDI distintos.

Adicionalmente a todos estos dispositivos, cabe destacar también la gran cantidad de herramientas software en el mercado que permiten secuenciar y sintetizar grabaciones MIDI. Ante la proliferación de este tipo de aplicaciones que hacen más accesible la producción de contenido musical, surge la necesidad cada vez más importante de que los distintos instrumentos MIDI que se fabrican, incorporen conexiones USB para poder conectar el dispositivo a un ordenador, permitiendo ampliar mediante software el número de controles a nuestra disposición sin necesidad de tener que adquirir un controlador hardware adicional, y con la absoluta flexibilidad que suelen dar este tipo de herramientas sin grandes complicaciones. También existen adaptadores USB-MIDI que permiten la conexión a un PC de dispositivos que sólo posean conectores MIDI. A parte del USB, también existen aparatos que utilizan una conexión WiFi para poder comunicarse con un ordenador.

Existen multitud de programas software para trabajar con MIDI. Anvil-Studio, Cubase, FL Studio, GarageBand, Digital Performer, MidiOX…etc. Las funcionalidades son diversas y pueden ir desde simplemente monitorizar los comandos MIDI entrantes, a auténticos secuenciadores software que permiten componer una grabación MIDI completa, entre muchas otras.

[2] http://es.yamaha.com/es/products/musical-instruments/strings/silentviolins/sv-150/?mode=model a fecha de Agosto de 2016
http://www.akaipro.com/product/ewi4000s a fecha de Agosto de 2016
https://www.roland.co.uk/products/td-4s/ a fecha de Agosto de 2016

3 MÓDULO BLUETOOTH HC-05

En este capítulo se llevará a cabo una descripción de la tecnología Bluetooth y se repasará brevemente el estado del arte en este campo. Además, se enumerarán las distintas características de los módulos Bluetooth usados en este proyecto para implementar la conexión inalámbrica entre controlador e instrumento y se justificará la elección de los mismos.

3.1 Tecnología Bluetooth

El Bluetooth surge como un estándar de comunicación inalámbrica principalmente orientado a comunicaciones a corta distancia, tras la unión y el acuerdo de diversas empresas tecnológicas como IBM, Nokia, Toshiba o Intel. Desde entonces, el estándar se ha ido mejorando ampliamente en sus sucesivas nuevas versiones, sin perder la compatibilidad con las anteriores.

Figura 3-1. Logotipo comercial Bluetooth[3]

En sus dos primeras versiones, "v1.0" y "v1.0B", el estándar sufrió de numerosos problemas que dificultaron en gran medida su implantación. Estos se corrigen en su mayoría con la versión "v1.2", la cual se ratifica como estándar IEEE. A partir, se añaden nuevas funcionalidades con las nuevas iteraciones y se mejora la velocidad de transmisión progresivamente. De esta manera, se pasa de los 721 kbps que se lograban en la versión "v1.2", a los 3 Mbps de la versión "v2.0 + EDR", hasta 24 Mbps en "v3.0+HS" al introducir el estándar 802.11 para transmisiones a alta velocidad, y finalmente 32 Mbps en "v4.0". Es en esta última, donde se bifurca por primera vez el estándar en su versión clásica orientada a conseguir la máxima velocidad posible de transmisión, y una nueva especificación del estándar llamada "Bluetooth Low Energy" orientada a conseguir el menor consumo disponible para el uso en sistemas con menores exigencias. [8] [9]

Los dispositivos Bluetooth transmiten información en las bandas ISM de radiofrecuencia de 2.4GHz. Según la potencia máxima de transmisión, pueden clasificarse en tres clases:

Clase	Potencia máx. (mW)	Potencia máx. (dBm)	Alcance (m)
Clase 1	100	20	100
Clase 2	2.5	4	5-10
Clase 3	1	0	1

Tabla 3-1. Clasificación dispositivos Bluetooth

[3] http://seeklogo.com/bluetooth-logo-20565.html# a fecha de Agosto de 2016

Para seguir el estándar, a todo dispositivo Bluetooth que sale al mercado se le asigna una dirección MAC única de 48 bits (6 bytes). Su formato en hexadecimal se basa en las tres componentes: NAP:UAP:LAP. La componente NAP (parte no significativa) está compuesta por 2 bytes y es asignada por el IEE a los distintos fabricantes. Por otro lado, la componente UAP de la dirección (parte alta) está compuesta por 1 byte, y la componente LAP (parte baja) por 3 bytes.

Además, es necesario que sean compatibles con algunos de los perfiles Bluetooth que se definen en el estándar. Estos perfiles, son los distintos servicios que un dispositivo Bluetooth puede llevar a cabo. Permiten unificar y estandarizar las diversas funcionalidades, para facilitar la compatibilidad entre dispostivos. Es importante a la hora de implementar una comunicación Bluetooth, que los dispositivos en cuestión posean los perfiles necesarios para llevar a cabo la tarea requerida. Algunos ejemplos de perfiles:

- Perfil A2DP. Permite la transmisión de un streaming audio (mono o estéreo) a través de la conexión Bluetooth.
- Perfil HFP. Ofrece servicios de manos libres para telefónos móviles. Normalmente audio en mono.
- Perfil HID. Da soporte a dispositivos de interfaz humana, como pueden ser ratones, teclados o controladores de videojuegos.
- Perfil HSP. Permite la conexión de auriculares Bluetooth.
- Pefil BIP. Permite el envío de imágenes sobre una conexión Bluetooth.
- Perfil BPP. Se utiliza para enviar documentos a impresoras.
- Perfil VDP. Streaming de video.
- Perfil SPP. El enlace Bluetooth emula una conexión serie entre los dos dispositivos para el intercambio de datos.

El Bluetooth, presenta como principal ventaja ante otras tecnologías como los infrarrojos, el hecho de que no sea necesario alinear los dispositivos que se comunican entre sí, e incluso pueden estar en habitaciones separadas, lo cual no es posible con infrarrojos al ser bloqueado por obstáculos, como la luz visible. Este hecho reduce enormemente el alcance práctico de los infrarrojos en comparación al Bluetooth.

Si se compara con una tecnología como WiFi, la gran ventaja es el bajo consumo del Bluetooth frente a la primera. Lo que justifica el uso de la tecnología Blutooth en aplicaciones menos exigentes. Si se demanda una mayor seguridad en la conexión, mayor rapidez, mayor distancia o crear una red amplia de propósito más general de dispositivos interconectados, puede ser más recomendable la tecnología WiFi.

3.2 Estado del Arte

De cara al futuro, la nueva versión de Bluetooth “v5.0” está ya anunciada para estrenarse a lo largo de 2017. Entre otras mejoras, se espera un aumento de la velocidad de transmisión, duplicando los valores actuales, así como llegar a un alcance cuatro veces mayor a los vigentes. [10] De esta manera, se prevee un salto cualitativo de esta tecnología para no perder terreno frente a la tecnología WiFi y resistir la aparición de otras tecnologías como el NFC. Por otro lado, siguen surgiendo nuevos perfiles que ofrecen nuevas y más avanzadas funcionalidades. Por ejemplo, el reciente perfil VCP para videoconferencias.

Con la aparición de nuevas actualizaciones del estándar, se mantiene la compatibilidad con los dispositivos antiguos, que en todo caso pueden seguir siendo una fenomenal opción si no se requieren las mejoras de las nuevas versiones, manteniendo la posibilidad de poder comunicarse con dispositivos de versiones posteriores.

En la actualidad, es innegable la gran extensión de esta tecnología en buena parte de los dispositivos electrónicos que pueden encontrarse en el mercado. A partir de la popularidad que fue granjeándose en los primeros telefónos móviles que incorporaron el Bluetooth como alternativa a los infrarrojos para compartir archivos, el crecimiento de esta tecnología ha sido tan notable, que es ya frecuente encontrarla en dispositivos muy variados, desde coches o electrodomésticos hasta auriculares inalámbricos. Todo esto provoca que el Bluetooth esté ya jugando un papel muy importante en el desarrollo del “Internet of Things”, sobre todo en situaciones donde es imprescindible un bajo coste, o estamos ante sistemas alimentados por batería y el

consumo es un aspecto muy relevante. De esta manera, cada vez más dispositivos IoT, hacen uso de Bluetooth para comunicarse localmente con otros dipositivos dentro del mismo espacio.

Así mismo, son numerosos los fabricantes que optan por esta tecnología para sacar al mercado productos inalámbricos. Por ejemplo, en el mercado de consolas, el Bluetooth ha sido el estándar elegido para desarrollar controladores inalámbricos, como ha sido el caso de Sony o Nintendo. También en el campo del audio, son numerosas las marcas que comercializan productos que hacen uso de una conexión Bluetooth para ofrecer altavoces y auriculares sin cables.

3.3 Módulo HC-05

El modulo HC-05 es un módulo Bluetooth de clase 2 y versión "v2.0 +EDR". Esto, como se vio en el primer apartado del capítulo, le confiere una potencia de transmission de 2.5 mW y un alcance entre 5 y 10 metros. Así mismo, la velocidad de transferencia tendrá su límite en torno a los 3 Mbps. Se alimenta a 3.3V, por lo que es compatible con los microcontroladores de Texas Instruments que se utilizarán en el proyecto. Por defecto funciona con el perfil SPP (Perfil de Puerto Serie Virtual), que es el adecuado para la tarea requerida. [4] [11]

Figura 3-2. Módulo HC-05

Es capaz de comunicarse con un microcontrolador a través de una UART. Así mismo, el modulo Bluetooth puede ser enlazado con otro modulo que se comunique a la misma velocidad y tenga la misma contraseña o con otros dispositivos que tengan un adaptador Bluetooth, como puede ser un teléfono móvil o un ordenador. Es importante tener en cuenta que el modulo puede jugar el papel de "master" o el de "slave" en una comunicación. Esto se ha de considerar a la hora de enlazar dos módulos, ya que ambos no pueden tener el mismo rol. El módulo posee un pequeño led de color rojo que parpadea rápidamente hasta que es enlazado con éxito a otro dispositivo, cuando empieza a parpadear a un ritmo más lento. Para conectarse por Bluetooth al adaptador de un telefóno móvil o un PC, es imprescindible que el módulo esté configurado como "slave" (configuración por defecto).

Existen dos versiones en el mercado de este módulo, entre las cuales varian levemente la distribución de los pines y aspectos de la configuración del mismo. En este caso, los módulos a usar disponen de 6 pines. Serán relevantes para este proyecto los pines de alimentación (VCC), tierra (GND), lectura UART (RXD) y escritura UART (TXD). Además, posee un botón que al ser pulsado prepara al módulo para recibir comandos AT.

Figura 3-3. Pines HC-05

A parte, el mismo fabricante del módulo HC-05 ofrece otra versión, la HC-06, también orientada a aplicaciones de electrónica de consumo. Es básicamente el mismo módulo con un firmware distinto. Sin embargo, el HC-06 presenta la desventaja de que no puede ser reconfigurado como "master" o "slave", por lo que es bastante menos flexible en este aspecto. [11]

En general, la elección de los módulos HC-05 para implementar la comunicación remota del proyecto, se ha basado en el hecho de ser módulos que aportan una gran flexibilidad al proyecto, con posibilidad de reconfigurarlos y que una vez son configurados correctamente, el funcionamiento es notablemente sencillo. Están además basados en un estándar con una fuerte presencia en el mundo industrial como es el Bluetooth, y su coste es bastante reducido.

3.4 Configuración y manejo del HC-05

El hecho de que el dispositivo actúe como "master" o "slave", así como otra serie de parámetros como puede ser el nombre del dispositivo, la contraseña o la velocidad de comunicación, es necesario configurarlo mediante el envío de comandos AT. Cuando un mensaje es recibido e interpretado correctamente como un comando AT (con el botón pulsado), el nuevo valor del parámetro queda grabado indefinidamente hasta que vuelva a ser modificado. Como por defecto el módulo viene a 9600 bps, se puede utilizar el MSP430G2553 con los jumpers en modo SWUART para enviar los distintos comandos AT necesarios conectando el Launchpad del MSP430 a un PC mediante un puerto USB. Se puede utilizar cualquier terminal software que permita enviar mensajes por el puerto serie, como puede ser el programa TeraTerm. El Launchpad debe aparecer como "MSP430 Application UART".

Es conveniente remarcar el hecho de que cada mensaje que envíemos debe acabar con "/r/n" (retorno de carro y salto de línea). Para mayor facilidad, se puede marcar en el terminal la opción de acabar los mensajes por defecto con CR+LF. Al mandar un comando AT, recibiremos una respuesta, que será OK + Información sobre el parámetro cambiado, o ERROR si no se recibió un comando reconocido por el dispositivo.

Es además necesario, que el módulo se encuentre enlazado a otro dispositivo para que la configuración funcione correctamente. Conociendo todo esto, se puede abordar la configuración del dispositivo. Como la necesidad en este caso es enlazar dos módulos, necesitamos básicamente asegurar que estos tienen iguales velocidades y contraseñas (por defecto la contraseña es 1234). También es necesario que uno tenga el rol de "master" y otro el de "slave" (por defecto es "slave"). A continuación, se presenta una tabla a modo de resumen de los comandos y parámetros más importantes [12]:

Nombre	Comando AT	Respuesta	Parámetro A	Parámetro B	Parámetro C
Test	AT	OK	—	—	—
Reset	AT+RESET	OK	—	—	—
Versión soft.	AT+VERSION?	+VERSION: <A> OK	Versión software	—	—
Parám. por defecto	AT+ORGL	OK	—	—	—
Obtener dirección Bluetooth	AT+ADDR?	+ADDR: <A> OK	Dirección Bluetooth	—	—
Obtener nombre	AT+NAME?	+NAME: <A> OK	Nombre dispositivo	—	—
Cambiar nombre	AT+NAME=<A>	OK	Nuevo nombre	—	—

Obtener rol	AT+ROLE?	+ROLE: <A> OK	Rol módulo: 0 - Slave 1 - Master 2 - Loop (slave)	—	—
Cambiar rol	AT+ROLE=<A>	OK	Nuevo rol	—	—
Obtener contraseña	AT+PSWD?	+PSWD: <A> OK	Contraseña dispositivo	—	—
Cambiar contraseña	AT+PSWD=<A>	OK	Nueva Contraseña	—	—
Obtener parámetros Comunicación serie	AT+UART?	+UART: <A>,<B>,<C> OK	Baud rate (bps)[4]	Stop Bit: 0 - 1 bit 1 - 2 bits	Bit paridad: 0 - No 1 - Impar 2 - Par
Cambiar parám. serie	AT+UART= <A>,<B>,<C>	OK	Nuevo baud Rate (bps)[4]	Nueva conf. Stop bit	Nueva conf. paridad
Obtener modo conexión	AT+CMODE?	+CMODE: <A> OK	Modo conex[5]: 0 – Conectar a dirección específica 1 – Conectar a cualquier dir. 2 - Loop (slave)	—	—
Cambiar modo conex.	AT+CMODE= <A>	OK	Nuevo modo de conexión	—	—
Obtener dirección Bluetooth de enlace	AT+BIND?	+BIND: <A> OK	Dirección[6] Bluetooth a la que enlazarse	—	—
Cambiar dirección Bluetooth de enlace	AT+BIND=<A>	OK	Nueva dirección	—	—

Tabla 3-2. Lista de comandos AT

[4] El baud rate (en decimal) tiene que ser alguno de los siguientes valores estándar: 4800, 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600, 1382400.

[5] Por defecto, el módulo se encuentra con modo de conexión 0. En este modo el módulo busca al dispositivo con la dirección Bluetooth especificada con el parámetro AT+BIND y se conecta a ese dispositivo y sólo ese. Para mayor sencillez y flexibilidad, está el modo 1, que se conecta a cualquier dispositivo en su radio de alcance que posea la misma contraseña y misma velocidad de transmisión.

[6] Siguiendo el formato explicado en anteriores apartados.

Con todos los parámetros que aparecen en la tabla 3-2, será suficiente para poder configurar los módulos HC-05 para que presenten el funcionamiento requerido para este proyecto. Tan sólo basta, como se explicó anteriormente en este apartado, conectar el Launchpad del MSP430 a un PC con un terminal como TeraTerm, cargar un proyecto vacío en el microcontrolador, poniendo los jumpers en SWUART, para utilizar la conexión USB y mandarle los comandos AT al módulo desde el terminal en el PC. Se ha de elegir conexión serie y el puerto que ponga "MSP430 Application UART". Para mayor comodidad, configurar en Setup para que envíe CR+LF al final de cada transmisión. Por último, se debe enlazar el módulo Bluetooth a otro dispositivo. Para probar que el funcionamiento es el correcto, se puede probar con el comando de test AT. Si no hay error, la respuesta será "OK".

El objetivo es que al menos uno de los módulos, que por defecto están en rol "slave", actúe como maestro de la comunicación. Para ello, en la configuración de alguno de ellos, envíamos el comando AT+ROLE=1. Si se desea, se puede cambiar el nombre de los dispositivos mediante AT+NAME. Posteriormente, se configurarán ambos para que el modo de conexión sea a cualquier dispositivo, sin especificar una dirección en especial. Esto es conveniente, porque el enlace funciona de manera correcta en este modo, sin necesidad de especificar direcciones que tendrían que cambiarse si se sustituyen alguno de los dos módulos. Envíamos el comando AT+CMODE=1. En este caso, no es necesario modificar nada con el comando AT+BIND, ya que, aunque hubiera alguna dirección guardada en el registro correspondiente, ésta se ignoraría al estar en el modo 1.

Es conveniente comprobar que la contraseña en ambos dispositivos es la misma. Por defecto suele ser la contraseña "1234". Para ello, podemos preguntar por la misma con el comando AT+PSWD?

Por último, resta configurar los parámetros de la comunicación serie. Es importante que la velocidad sea la misma en ambos para que puedan enlazarse sin problemas. Es necesario remarcar, que ambos módulos trabajarán con perfil SPP, simulando una línea serie, que básicamente traslada los comandos MIDI generados en el emisor hasta el receptor. Es importante que el receptor se comunique a los 31 250 bps característicos del estándar MIDI con el instrumento a controlar. Sin embargo, en la comunicación intermedia entre emisor y receptor esto no se tiene que cumplir. De hecho, este valor de la velocidad de transmisión no se encuentra entre los valores estándar que se pueden adjudicar a los módulos HC-05 mediante el comando AT+UART. Teniendo además en cuenta que el protocolo Bluetooth implica el envío de una serie de cabeceras extra a los datos que se transmiten, es conveniente que la velocidad elegida sea mayor. Por estos motivos, se configurarán los módulos para que trabajen a una velocidad de 115 200 bps. Para ello, envíamos el comando: AT+UART = 115200,0,0. Mediante los otros dos parámetos, se configura la transmisión serie a trabajar con un solo bit de stop y sin bit de paridad, de cara a seguir el formato de un mensaje MIDI.

Es necesario destacar el hecho de que todos los parámetros que cambiemos, no se verán formalmente reflejados en el funcionamiento del módulo hasta que se quite la alimentación. Esto es importante porque la conexión serie del Launchpad está capada para trabajar sólo a 9 600 bps, por lo que cuando el cambio en la velocidad se refleje se manera definitiva, será necesario seguir otro método distinto para reprogramar el módulo en cuestión. Mientras no se elimine la alimentación del dispositivo, se puede seguir enviando comandos AT para programar otros parámetros, ya que la velocidad efectiva seguirá siendo la misma, incluso aunque al preguntar mediante AT+UART? salga que la velocidad es ya de 115 200 bps.

Finalizada la programación de los módulos, todo está preparado para que ambos se enlacen de manera automática si están alimentados. La conexión con los microcontroladores del bloque emisor y receptor, y la programación necesaria para el manejo de los mismos desde los MSP430, se detallarán en los próximos capítulos del proyecto.

4 PANTALLA VM800

En este capítulo se hará un repaso de la evolución de las pantallas táctiles, en especial las de tecnología resistiva. Se abordarán las características de estas pantallas, y se describirá detalladamente la pantalla VM800, elegida para implementar la interfaz del controlador MIDI.

4.1 Pantallas táctiles

Desde la aparición de los primeros sistemas electrónicos, tanto en el campo industrial como en el de consumo, ha sido necesario idear sistemas para que el usuario pueda interactuar con estos sistemas e introducir consignas. Botones, potenciómetros o teclados, todos ellos tienen el mismo objetivo: dotar al sistema electrónico de una interfaz mediante la cual poder utilizarlos y manipular su funcionamiento.

Sin embargo, ante el incremento de la complejidad de los distintos procesos en la industria, estos sistemas de control electrónicos son, a su vez cada vez más complejos, implicando la necesidad de ofrecer al operario a cargo de esa máquina un mayor número de controles con los que ajustar todos los parámetros y tener un control adecuado a la complejidad de esa máquina. Ante esta situación, era evidente que los clásicos sistemas de interacción del usuario empezaban a no ser adecuados. Por un lado, se complica el diseño y la operación de un panel de control. Por otro lado, disminuye la robustez de los mismos ante la dependencia de tantos elementos con componentes mecánicos que pueden fallar habitualmente.

Por estas razones, se desarrollan los primeros sensores y pantallas táctiles en la industria. Con el objetivo, de simplificar los diseños, aumentar la robustez de los mismos, facilitar el mantenimiento y ofrecer un control más simple y accesible. Adicionalmente, las pantallas táctiles no sólo significaban un cauce de entrada de datos al sistema, si no que al mismo tiempo actúan como salida del mismo, mostrando los resultados de las distintas operaciones. Fue posteriormente, cuando empiezan a verse los primeros sensores táctiles en dispositivos de electrónica de consumo. La idea era hacer a estos aparatos mucho más fiables, intuitivos y accesibles para el usuario. De esta manera, surgen nuevos dispositivos electrónicos que basaron su éxito comercial en ofrecer una interacción mucho más sencilla y novedosa para el gran público, mediante el equipamiento de sensores y de pantallas táctiles.

Existen básicamente, dos grandes tipos de pantallas táctiles en el mercado. Por un lado, están las pantallas resistivas (las primeras en surgir), y por otro lado están las pantallas capacitivas.

Las pantallas resistivas, se basan en la superposición de dos capas de material elástico resistivo (resistividad en el orden de los 100Ω). Tiene cuatro contactos metálicos o conexiones, dos para situar la pulsación a lo largo del eje X (X+, X-), en una de las capas, y otras dos para situarla a lo largo del eje Y (Y+, Y-), en la otra capa.

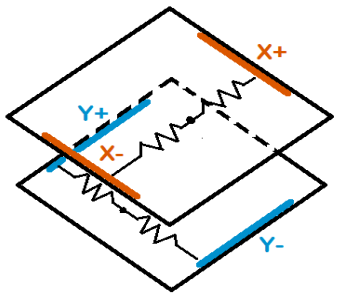

Figura 4-1. Capas pantalla resistiva

El principio de funcionamiento se basa en poner uno de los contactores de un eje como entrada digital a nivel lógico alto con una resistencia de pull-up (debe de tener un valor de al menos un orden de magnitud mayor que la resistencia de la capa resistiva). El contactor contrario se deja abierto. En el otro eje, que está en una capa distinta, uno de los contactores se pone a tierra y el contrario se deja abierto. De esta manera no circula corriente por ninguna de las ramas ya que ambas están en circuito abierto. Esto se puede conocer porque el contactor con el pull-up permanece a tensión Vcc. En caso de que se produzca una pulsación, se produce un contacto entre ambas placas que origina un camino de baja impedancia a tierra para que se conduzca corriente. El valor de la entrada digital será ahora un 0, pudiendo de esta manera detectar la pulsación. [4]

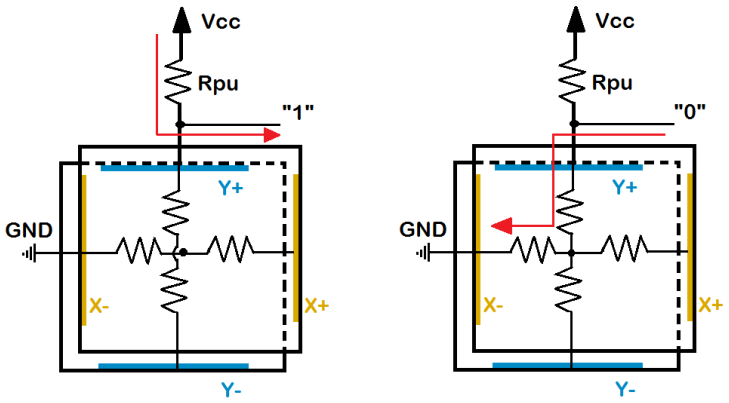

Figura 4-2. Pantalla resistiva sin pulsar y pulsando

Para conocer la posición exacta de la pulsación, es necesario hacer dos medidas distintas, una para el eje X y otra para el eje Y. Para la medida del eje Y, se pone un extremo del mismo a Vcc y el otro a tierra. En el eje X, un extremo se deja sin conectar y el otro entra en un ADC que lleva a cabo la medida de la tensión.

Contra mayor sea la medida de tensión, en una posición más elevada del eje Y nos encontramos. De esta manera, la posición en el eje Y vendría dada por el valor:

$$Pos_y = \frac{V_x}{V_{OH}} Y_{max}$$

El proceso para obtener la medida para el eje X es similar, y la posición en el mismo se obtendría:

$$Pos_x = \frac{V_y}{V_{OH}} X_{max}$$

Debido a la naturaleza de este tipo de pantallas, se puede inferir que la precisión de las mismas va a depender enormemente de la calibración del valor de tensión V_{OH} y del ADC. Adicionalmente, al tener que superponer varias capas, el brillo será más reducido que en otras tecnologías como las pantallas capacitivas. Son sólo capaces de detectar un punto al mismo tiempo y es necesario ejercer una presión considerable. Como grandes ventajas, se encuentra el hecho de que es una tecnología bastante barata de implementar, y que pueden ser utilizadas con guantes o un puntero. Son más resistentes a agentes externos como el polvo y agua.

Las pantallas capacitivas, por otro lado, basan su funcionamiento en la capacidad característica de un dedo o la mano de una persona. Al tocar la pantalla con un dedo, la capacidad parásita del mismo hace variar la capacidad del sensor capacitivo correspondiente, de manera que un simple circuito oscilador R-C vería afectada la frecuencia de oscilación en la señal de salida con la variación de esa capacidad. Mediante una matriz de sensores capacitivos, se puede llevar a cabo una pantalla táctil capacitiva. Debido a esto, el coste de una pantalla capacitiva aumenta notablemente al aumentar el número de puntos (precisión) de la misma, resultando, por tanto, una tecnología por lo general más cara que las resistivas.

Pese a todo esto, las pantallas capacitivas han experimentado un crecimiento enorme en los últimos años relegando las pantallas resisitivas a una posición muy menor en el mercado. Esto se debe a la mejor respuesta de las pantallas capacitivas, donde la presión a realizar es prácticamente despreciable. El hecho de que no sea necesario apilar varias capas les confiere mayor brillo y mejor calidad de imagen. Y, por encima de todo, permiten la detección simultánea de varias pulsaciones (multitouch).

4.2 Pantalla VM800

De cara a cumplir los objetivos del proyecto, se podría haber optado por un diseño más convencional de controladores físicos mediante pulsadores o potenciómetros. Sin embargo, se optó en este caso por el uso de una pantalla táctil resistiva, de cara a ofrecer un control más intuitivo, y en el que todo momento se pudiera conocer el estado de los distintos controles de los diferentes canales, dentro de una interfaz de tamaño muy reducida. Mediante una pantalla, se pueden programar diversos menús con distintos controladores y guardar en memoria la posición de los mismos. Con controladores físicos, o se aumenta el tamaño considerablemente para poner todos los controladores necesarios, o se tendrían que asociar varios comandos a uno solo, sin posibilidad de mantenerlo en el valor actual si es necesario actuar sobre otra funcionalidad del mismo. Adicionalmente, se evita el uso de elementos mecánicos que siempre afectarán negativamente a la fiabilidad del sistema, en contraposición a la robustez de un control con una pantalla táctil. Por el contra, aumenta la complejidad del código en el bloque emisor que se comunica con la pantalla, y el tamaño del mismo.

En el primer caso, se cuentan con librerías de funciones para la pantalla, que han sido realizadas por el profesor Manuel Ángel Perales Esteve para su uso en asignaturas y proyectos del departamento de Ingeniería Electrónica. Será, sin embargo, necesario llevar a cabo modificaciones para añadir nuevas funcionalidades, y para llevar a cabo una nueva distribución de los pines utilizados para poder utilizar simultáneamente tanto la pantalla como el módulo Bluetooth.

Para el segundo caso, es necesario respetar la capacidad el microcontrolador y conseguir que el código tenga un tamaño lo bastante reducido como para poder ser grabado en la memoria del mismo. Para ello, fue necesario implementar un controlador de 4 canales, de los 16 disponibles en MIDI, y elegir un número reducido de parámetros a controlar.

La pantalla a utilizar, será la pantalla VM800, del fabricante FTDI. Su principal característica, y que la hace ideal para este proyecto, es su coprocesador gráfico, que permite la representación de elementos complejos en pantalla, sin necesidad de exigir demasiada potencia al microcontrolador que gestionará la pantalla. De este modo, tenemos a nuestra disposición la posibilidad de hacer diseños bastante ambiciosos que de otra manera serían imposibles de alcanzar con un microcontrolador como el MSP430G2553. Estamos, además, ante una pantalla táctil resistiva que permitirá interactuar con estos elementos en la propia pantalla. Ésta, además, cuenta con una resolución de 320x240 y 256k colores. [4]

Figura 4-3. Pantalla VM800[7]

La conexión con el microcontrolador, se realiza mediante comunicación SPI. Mediante esta conexión, se establece una comunicación serie y síncrona (a parte de línea de datos, línea de sincronismo), con la que se pueden enviar los comandos y datos necesarios para la representación en pantalla.

[7] http://www.ftdichip.com/Products/Modules/VM800B.html a fecha de Agosto de 2016

La comunicación SPI, implementa además un protocolo "master" "slave" en el que el dispositivo maestro inicia siempre la comunicación. Mientras que se produce la escritura de datos, se lleva a cabo la lectura al mismo tiempo. De esta manera, existen al menos 4 líneas necesarias. Por un lado, la línea de reloj (CLK), que envía el maestro para establecer la comunicación, la línea de "Chip Select" (CS), mediante la cual el maestro elige el dispostivo esclavo con el que se comunicará, y las dos líneas de datos: MOSI y MISO. Éstas son respectivamente, las líneas de salida del "master" y de entrada de datos del mismo. Correspondiendo igualmente, con la entrada al dispositivo esclavo y la salida. Se llevará a cabo un análisis más exhaustivo y particularizando para la programación del SPI del MSP430, en próximos apartados.

La interpretación de estos comandos que llegan a la pantalla corre a cargo del procesador gráfico FT800. Éste permite que no sea necesario refrescar la información contastemente desde el MSP430 para mantener lo dibujado sobre la pantalla, si no que, tras el envío de un comando para dibujar un objeto, este se mantiene en pantalla sin necesidad de volver a enviar el comando.

Como principales características, el FT800, que pertenece a la familia de sistemas EVE (Embedded Video Engine) de FTDI, posee una velocidad de reloj de hasta 48MHz, así como una memoria interna para objetos de hasta 256 kBytes. Posee tanto interfaz SPI de hasta 30MHz, como I^2C de hasta 3.4MHz para comunicarse con un microcontrolador maestro. Existe una pequeña excepción, y es que para activar el reloj del sistema, es necesario seguir una serie de pasos, durante los cuales es conveniente mantener el reloj del maestro por debajo de 11 MHz. Una vez terminado, sí se puede subir hasta 30 MHz. [13]

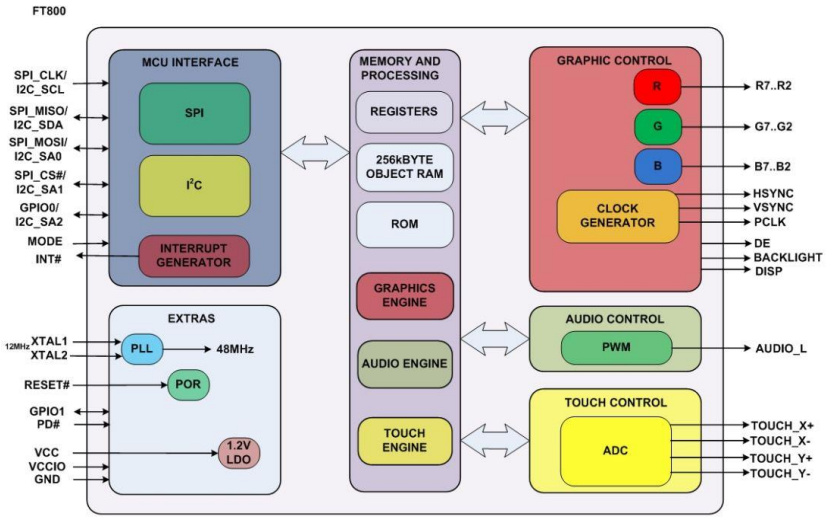

Figura 4-4. Diagrama de bloques del FT800[8]

Como se puede observar en el diagrama de bloques del FT800, posee tres grandes bloques de procesamiento, dedicados al control de los gráficos, con salida RGB de ancho 6 bits, al audio, con un sintetizador basado en MIDI integrado, y a la lectura de los toques en la pantalla, para la cual requiere un ADC para la lectura de los ejes X e Y, como se explicó en el anterior apartado.

La pantalla, puede ser alimentada con los 3.3V propios del MSP430G2553. Posee un regulador interno que permite alimentar el núcleo del procesador a 1.2V.

Todos los registros del FT800 están formados por 3 bytes. De estos, los dos bits MSB están reservados a proporcionar información respecto a la interfaz de comunicación con el "master". Un valor de "00" indica lectura en la dirección de memoria indicada por los 22 bits restantes. Un valor de "10" indica escritura, "01" se reserva para comandos del maestro o host ("Host Commands") y "11" queda indefinido. [13] El mapa de memoria del dispositivo es el siguiente:

[8] http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT800.pdf a fecha de Agosto de 2016 (Datasheet FT800)

Start Address	End Address	Size	NAME	Description
00 0000h	03 FFFFh	256 kB	RAM_G	Main graphics RAM
0C 0000h	0C 0003h	4 B	ROM_CHIPID	FT800 chip identification and revision information: Byte [0:1] Chip ID: "0800" Byte [2:3] Version ID: "0100"
0B B23Ch	0F FFFBh	275 kB	ROM_FONT	Font table and bitmap
0F FFFCh	0F FFFFh	4 B	ROM_FONT_ADDR	Font table pointer address
10 0000h	10 1FFFh	8 kB	RAM_DL	Display List RAM
10 2000h	10 23FFh	1 kB	RAM_PAL	Palette RAM
10 2400h	10 257Fh	380 B	REG_*	Registers
10 8000 h	10 8FFFh	4 kB	RAM_CMD	Command Buffer

Tabla 4-1. Mapa de memoria FT800[9]

Más allá de la dirección 0x108FFF, son registros reservados por el dispositivo. Se puede observar, como a parte de los 256kB de memoria para objetos, se disponen 275kB para fuentes y bitmaps o 380 bytes para registros de hasta 32 bits de diversa índole que sirven para configurar ciertos aspectos, como por ejemplo la frecuencia del reloj, efectos de sonido, calibración de la pantalla…etc. Tienen permiso de lectura, escritura o ambos. Por último, se reservan 4kB de memoria para los comandos que se reciban para objetos de pantalla.

De esta manera, a través del SPI, podemos hacer tres operaciones básicas desde el microcontralador maestro: Lectura de un registro en memoria, escritura, y envío de "Host commands". Para apuntar a la dirección necesaria, siempre empezar con los dos bits requeridos en función de la operación, como se explicó anteriormente. Por ejemplo, para una lectura en la dirección "0x102400", el valor de la dirección a enviar será éste mismo porque empieza con dos bits a cero. Si fuera escritura, sería necesario enviar "0x902400". El registro en memoria será exactamente el mismo, sólo que los dos primeros bits son "10" e indican escritura. En todo caso, los datos siempre enviados desde el MSB hasta el LSB.

Para comenzar la comunicación, se pone el "Chip Select" a nivel bajo, y sólo cuando ésta termina, se vuelve a poner a nivel alto. Entre medio, será siempre necesario comunicar primero la dirección de memoria sobre la que se actuará (siempre por la línea MOSI), y segundo llevar a cabo el transvase del dato a leer o escribir en memoria. En caso de hacer una lectura de un registro, se recibe por la línea MISO. Para escribir, se envía por la línea MOSI. En todo caso, siempre que se lea un registro, una vez se envíen los 3 bytes de la dirección, es necesario enviar un último byte de ceros, lo que se conoce como un "dummy" byte.

Para enviar un comando de host, se procede de manera similar a la lectura/escritura de registros. Se envían los 3 bytes correspondientes, conteniendo el primer byte el código de 6 bits del comando en sus bits LSB. Los dos bits MSB han de ser "01" para indicar el comando.

Para el envío de una lista de comandos para pintar una pantalla (no confundir con "Host commands"), se dispone de una cola circular tipo FIFO de 4kB que va desde la dirección de memoria "0x108000" hasta "0x108FFF". Los nuevos comandos se van escribiendo en la posición REG_CMD_WRITE (tendrá un offset respecto de la posición inicial, que tendrá que ser calculado por el maestro para cada comando), y ejecutan por REG_CMD_READ. [14] Es necesario que ocupen un múltiplo de 4 bytes. Estos comandos, son diversos y su función suele ser básicamente indicar la representación de un objeto, con una serie de parámetros determinados, en una posición de la pantalla. En algunos casos, se permite la interacción con los mismos.

[9] http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT800.pdf a fecha de Agosto de 2016 (Datasheet FT800)

Algunos ejemplos son [15]:

- CMD_TEXT: Dibujar texto.
- CMD_BUTTON: Dibujar botón.
- CMD_CLOCK: Dibujar reloj analógico.
- CMD_PROGRESS: Dibujar barra de progreso.
- CMD_SLIDER: Dibujar barra deslizadora.
- CMD_DIAL: Dibujar un control rotatorio.
- CMD_SCROLLBAR: Dibujar barra de scroll.
- CMD_TOGGLE: Dibujar selector de dos posiciones.
- CMD_KEYS: Dibujar fila de teclas (botones sin separación).
- CMD_GAUGE: Dibujar marcador de aguja.
- CMD_FGCOLOR: Cambiar color de primer plano.
- CMD_BGCOLOR: Cambiar color de fondo.
- CMD_GRADIENT: Dibujar gradiente entre dos colores.
- CMD_REGREAD: Leer registro indicado.
- CMD_MEMWRITE: Escribir en el registro indicado.

Para indicar el comando a utilizar, se envía como dato (tras el envío de la dirección adecuada) el código de 4 bytes asociado al comando (para conocerlo es necesario consultar la guía de programador aportada por el fabricante para el FT800. Después de este comando, se envían los datos asociados. Por ejemplo, el comando CMD_BUTTON, es el comando "0xFFFFFF0D". Se enviarían como primer dato esos 4 bytes. A continuación, se ha de envíar la coordenada X y la coordenada Y. Ambos ocupan 2 bytes cada uno. Posteriormente, es necesario indicar la anchura y la altura del botón, parámetros que también son de 16 bits. Después, se han de rellenar otros dos parámetros de 2 bytes, uno con la fuente a usar para los caracteres que se quieran escribir dentro del botón, y otro que es un valor que sirve para configurar algunas opciones del objeto. En este caso, sirve para indicar si el botón se ha de dibujar con efecto 3D (valor 0), o con efecto de pulsación (valor 256). Por último, se envía la cadena de caracteres a mostrar dentro del botón.

Si se hacen cuentas, se puede observar que sin contar la cadena, se están enviando 16 bytes, que es múltiplo de 4. Si no se quisiera representar nada, y no se enviase una cadena, no sería necesario hacer nada más. En cuanto se envíe un solo carácter a representar, es necesario comprobar que el número resultante de bytes enviados sigue siendo múltiplo de 4, y si no es así, envíar los "dummy" bytes necesarios hasta conseguirlo.

De cara a la programación del dispositivo, es conveniente hacer funciones de alto nivel que implementen todo el proceso descrito de manera autómatica, siendo tan solo necesario introducir los parámetros requeridos, para cada uno de los comandos. Adicionalmente, también es recomendable utilizar funciones que automaticen los procesos de escribir o leer en memoria que también se han descrito anteriormente. En este caso, se parte para el proyecto, de la ya mencionada librería adaptada de un ejemplo del fabricante FTDI para otro microcontrolador, por el profesor Manuel Ángel Perales Esteve. Los cambios y ampliaciones realizados se detallarán en capítulos posteriores, donde se verán también estos comandos de manera más detallada.

Un programa tipo para el manejo de la pantalla, ha de contener una primera fase de configuración e inicialización de la pantalla, previo a un bucle donde se dispensan los comandos para pintar la pantalla, escribiéndolos en la FIFO circular de 4kB, en la posición adecuada. Posteriormente, se ordena la ejecución de estos comandos, que básicamente implica que el coprocesador gráfico pase estos comandos de la FIFO circular a la zona de memoria "RAM_DL", donde guarda la "Display List" de la pantalla a dibujar.

Para la conexión física del dispositivo, tenemos los siguientes pines, que han de ser asociados de manera adecuada al microcontrolador MSP430, teniendo en cuenta las funciones que se multiplexan en los mismos y que es necesario también conectar el módulo HC-05:

Nombre pin	Número	Descripción
SCLK	1	Señal de reloj SPI. Suministrada por el master
MOSI	2	Línea Master Out Slave In. Entrada de datos a la VM800
MISO	3	Línea Master In Slave Out. Salida de datos de la VM800
CS#	4	Chip Select. Permite al master elegir al esclavo con el que comunicarse
INT#	5	Salida de interrupciones al microcontrolador maestro
PD#	6	Control de Power Down
5V	7	Alimentación a 5V
3.3V	8	Alimentación a 3.3V
GND	9	Tierra
GND	10	Tierra

Tabla 4-2. Pines de la pantalla VM800

Los pines CS#, INT# y PD#, son activos a nivel bajo.

5 MICROCONTROLADOR MSP430G2553

A lo largo de este capítulo, se abordará en primera instancia algunas propiedades básicas de los microcontroladores y se hará un breve repaso de las distintas opciones que hay en el mercado. Se justificará la elección del sistema elegido y se detallarán las características del microcontrolador MSP430G2553 en el que se desarrollará el mismo.

5.1 Microcontroladores

Existen diversas opciones a la hora de desarrollar un sistema electrónico digital. En este caso, teniendo en cuenta las exigencias y necesidades del proyecto, parece inmediato decantarse por un sistema microcontrolador.

Un microcontrolador, es un circuito integrado programable pensado para ejecutar una serie de tareas programadas por el usuario. En contraposición a otros sistemas como los microprocesadores, suelen ser dispositivos de menor capacidad de cálculo que están orientados al más bajo consumo posible. Como grandes ventajas, suelen incorporar numerosos periféricos en su interior que le permiten abordar diversas tareas con una mayor integración, y se pueden encontrar dispositivos de este tipo desde muy bajo coste, permitiendo desarrollar proyectos de baja exigencia de procesamiento sin necesidad de acometer un gran gasto. Es frecuente que incluso en sistemas microprocesador más complejos, existan diversos microcontroladores destinados a gestionar otras tareas secundarias.

A la hora de apostar por un microcontrolador en particular, es necesario atender a sus especificaciones y comprobar que éstas serán suficientes para acometer las tareas requeridas, dentro de un precio acorde. Algunas de las características más importantes son:

- Memoria de programa y memoria RAM. La primera suele estar implementada con memorias no volátiles de tipo FLASH o EEPROM. Siendo la primera la más frecuente en la actualidad. Su principal cometido es contener el conjunto de instrucciones que conforman el programa. La memoria RAM, tiene como principal función almacenar información temporal necesaria para la ejecución del programa. Suelen implementarse con SRAM (estática), ya que la baja capacidad por superficie de ésta no es un gran impedimento al ser dispositivos con poca memoria RAM. Por contra, se evita la necesidad de refresco de las memorias dinámicas, mejorando en velocidad. En ambos casos, se suelen encontrar integradas en el chip del microcontrolador. [16]
- Capacidad de cómputo. Siempre es necesario considerar la complejidad de la solución a implementar para elegir un microcontrolador que posea la capacidad de cálculo suficiente. Para muchas aplicaciones, un microcontrolador de 8 bits puede ser suficiente, permitiendo ahorrar frente a opciones superiores. También es un parámetro a tener en cuenta, la velocidad de reloj del sistema. Normalmente los microcontroladores suelen trabajar en el rango de las decenas o centenares de MHz. Suelen implementar una aritmética de punto fijo, aunque existen algunos con aritmética de punto flotante.
- Periféricos. Una de las grandes ventajas de optar por un sistema microcontrolador es el hecho de contar dentro del mismo chip con numerosos periféricos que implementan diversas tareas y dotan al microcontrolador de capacidad para comunicarse con el exterior. Los periféricos que se pueden encontrar son diversos y varian enormemente entre una gama de precios u otra, ya sea en su presencia o no, el número de ellos y la potencia y capacidad de los mismos.

Algunos ejemplos de periféricos que se pueden encontrar en un microcontrolador son:

- Entrada/Salida de propósito general. Son la manera más básica que posee un microcontrolador para comunicarse con el exterior. De esta manera, se suelen poner a disposición del usuario una serie de pines cuya función más básica suele ser la lectura / escritura de de valores digitales. Suelen utilizarse para leer sensores digitales o actuar sobre ciertos dispositivos con valores binarios. Dependiendo del microcontrolador, suelen tener una serie de características relacionadas con la corriente admisible, protección de sobreintensidades y capacidad de programar interrupciones o resistencias de pull up / pull down. Debido al reducido tamaño, es frecuente que el número de pines disponible sea reducido, por lo que las funciones de E/S de multiplexan con otras distintas.
- Conversores Analógico/Digital. Es frecuente que la necesidad de interacción con elementos exteriores vaya más alla de leer o escribir valores digitales. Para poder gestionar señales analógicas, es frecuente dotar a los microcontroladores de conversores analógico/digital. Es mucho más frecuente ver conversores ADC para leer señales de sensores analógicas, que conversores DAC para convertir valores digitales a analógicos, siendo posible encontrar microcontroladores con ADCs desde gamas muy bajas. Será necesario tener en cuenta parámetros como la resolución de los mismos (normalmente para aplicaciones sencillas 8 ó 10 bits serán suficientes), velocidad de muestreo, rango de entrada, relación señal-ruido…etc.
- Temporizadores y Contadores. Poseen la función de sincronizar y medir la duración de eventos y dar una referencia de tiempo. Es conveniente tener en cuenta la anchura de bits de los mismos, que define el valor máximo de pulsos que se pueden contar, así como los modos de configuración que ofrezca el microcontrolador para poder usar los temporizadores y contadores para diversas tareas.
- PWM. Mediante la modulación por ancho de pulsos, se pueden efectuar numerosas tareas, como el control de servomotores, o implementar sencillos DACs.
- Módulos de comunicación. Sin duda, una de las características más importante es la capacidad de establecer protocolos de comunicación con otros dispositivos. Para ello, se suelen integrar módulos que permiten ser configurados para llevar a cabo transferencia de información con distitintos protocolos. Lo más habitual, es encontrar módulos de comunicación asíncrona (UART) y síncrona (I^2C, SPI).

A parte de toda esta gama de periféricos, y debido a la gran focalización de los microcontroladores en el bajo consumo, la gran mayoría de microcontroladores en la actualidad, suelen ofrecer la posibilidad de configurar modos de bajo consumo, especialmente útiles cuando se espera la llegada de una interrupción sin realizar en la espera ninguna tarea exigente. Esto es importante, de cara al funcionamiento del microcontrolador tras la programación del mismo, especialmente si va a ser alimentado con un sistema de baterías independiente.

Por último, existen siempre otra clase de factores que son importantes a la hora de decantarse por una opción del mercado u otra. Factores relativos al material y las herramientas de desarrollo que aporta el fabricante, y a la extensión o futuro crecimiento de la plataforma a elegir.

Las posibilidades que ofrece el mercado son extensas. Existen numerosos fabricantes de microcontroladores con diversidad de gamas y precios que buscan ofrecer un amplio abanico de alternativas. Algunos ejemplos son Freescale, con microcontroladores que van desde los 8 hasta los 32 bits, con familias como la 68HC08 ó 68HC11 entre otras. También el fabricante Microchip, con su extensa familia de controladores PIC, siendo uno de los primeros fabricantes en encontrar un notable éxito con productos orientados a principiantes y aficionados a la electrónica. Por supuesto, la plataforma Arduino, con una gran presencia en la actualidad en ese mismo sector, basada en microcontroladores de la familia AVR del fabricante ATMEL (aunque en la actualidad se fabrican placas Arduino con microcontroladores de otros fabricantes). También Texas Instruments, con sus numerosas gamas de MSPs (Mixed Signal Processor) y DSPs (Digital Signal Processor), entre las cuales se encuentran desde microcontroladores de reducido precio para aplicaciones simples y competir con Arduino, hasta aplicaciones industriales para el control de motores y procesado de audio y video.

5.2 Características MSP430G2553

Como se desprende del apartado anterior, las posibilidades en el mercado son numerosas. En este caso, para poder abarcar las pretensiones del proyecto, no son necesarias grandes capacidades de cálculo o una gran cantidad de memoria. Sí es fundamental que el microcontrolador elegido ofrezca módulos de comunicación SPI y UART, y que sea posible usarlos simultáneamente. El SPI es necesario para comunicarse con la pantalla, mientras que UART es requerida para los módulos Bluetooth y la comunicación MIDI.

Considerando todo esto, se opta por el microcontrolador MSP430G2553 de la familia MSP430 de microcontroladores de Texas Instruments. Es un microcontrolador con una gran disponibilidad en el departamento, de muy bajo coste (por debajo de los 10 euros con placa de desarrollo), y con la herramienta software de desarrollo, Code Composer Studio, que Texas Instruments ofrece gratuitamente para códigos por debajo de los 16K (que es justamente la capacidad del dispositivo en cuestión). Existe, además, una alta compatibilidad con otros microcontroladores de la misma familia.

Figura 5-1. MSP430G2553 en placa de desarrollo Launchpad[10]

Este microcontrolador, está basado en una CPU de 16 bits con arquitectura de Von Neumann y conjunto de instrucciones reducidas RISC, que implementa un conjunto de instrucciones más pequeñas y simples en pos de una mayor velocidad en la ejecución de las tareas, en contraposición a otros conjuntos de instrucciones más complejos, pero que puedan ofrecer una mayor versatilidad. En total, se contabilizan 27 instrucciones con 24 más emuladas. Así mismo, es capaz de trabajar hasta una velocidad de 16 MHz, permitiendo diversas configuraciones del reloj y fuentes externas. En cuanto a memoria, se disponen de 16 kB de memoria de programa (memoria flash) y 512 bytes de memoria RAM. [17]

Perteneciente a la familia de ultra bajo consumo "MSP430G2xxx", el microcontrolador nos ofrece 6 modos de funcionamiento entre los que se encuentra el modo normal de operación y 5 modos de bajo consumo que desactivan ciertas características para conseguir una reducción drástica del consumo. De manera muy aproximada, el consumo se reduce desde los 230 µA en modo normal a unos 0.5 µA en bajo consumo. Tarda menos de 1 µs en volver al modo normal de funcionamiento desde "standby". Para resumir estos modos:

- Active. Es el modo normal de funcionamiento, sin periféricos desactivados.
- Low-Power Mode 0 (LPM0). Desactiva: CPU, MCLK.
- Low-Power Mode 1 (LPM1). Desactiva: CPU, MCLK, DCO (si no se usa en modo normal).
- Low-Power Mode 2 (LPM2). Desactiva: CPU, MCLK, SMCLK.
- Low-Power Mode 3 (LPM3). Desactiva: CPU, MCLK, SMCLK, DCO.
- Low-Power Mode 4 (LPM4). Desactiva: CPU, MCLK, SMCLK, DCO, ACLK, oscilador.

[10] http://www.ti.com/ww/en/launchpad/launchpads-msp430-msp-exp430g2.html a fecha de Agosto de 2016

Como metodología general, se suelen invocar los distintos modos de bajo consumo al activar las interrupciones, de cara a no tener un consumo elevado mientras se espera la activación de las mismas para interrumpir la rutina de ejecución.

Existen tres formatos distintos en los que se puede adquirir este microcontrolador: PDIP, TSSOP y QFN. En este caso, se utilizará el microcontrolador con formato PDIP de 20 pines. Éste se caracteriza por tener el siguiente pin-out:

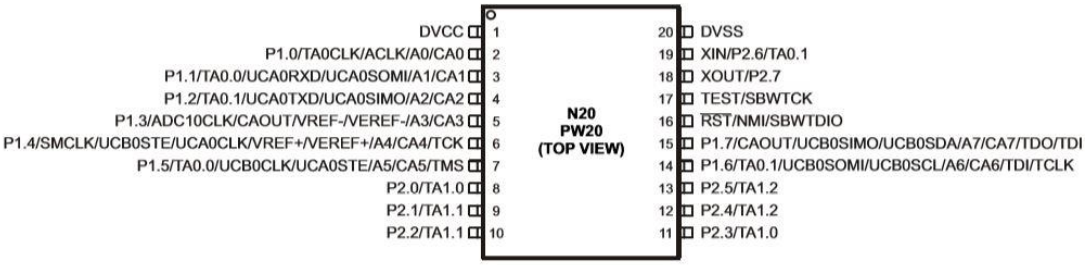

Figura 5-2. Pin-out del MSP430G2553 formato PDIP[11]

Como se puede observar, la gran mayoría de pines tienen multiplexadas numerosas funciones para conseguir una mayor integración del chip. Esto provoca que sea necesario programar los registros asociados a cada uno de ellos para conseguir la funcionalidad requerida, normalmente asociada a alguno de los periféricos que incluye el MSP430G2553. Anteriormente, ya se repasó resumidamente los periféricos que suelen albergar los microcontroladores. En el caso del MSP430G2553, está compuesto internamente por los siguientes periféricos:

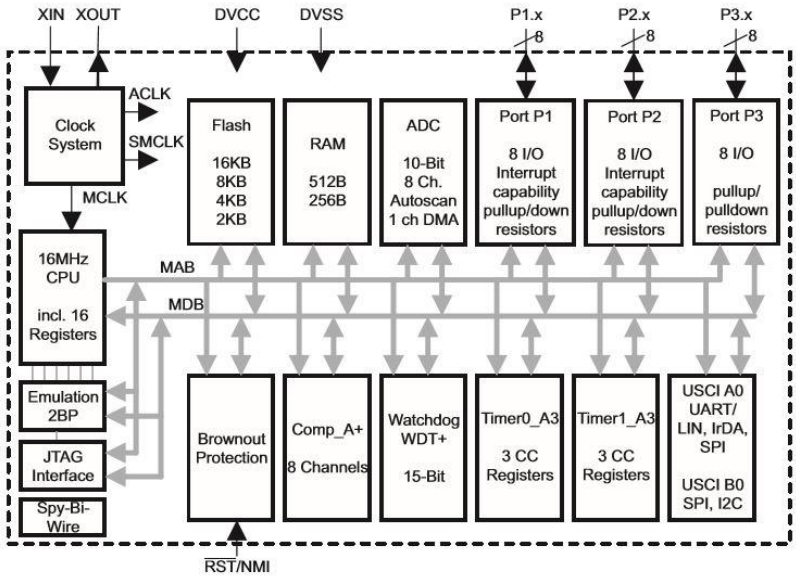

Figura 5-3. Diagrama de bloques del MSP430G2553[11]

Algunos de estos bloques han sido ya repasados anteriormente, como la memoria y la CPU. A continuación, se repasarán el resto de periféricos que componen el microcontrolador, con especial énfasis a los que serán utilizados en este proyecto.

[11] http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=msp430g2253&fileType=pdf a fecha de Agosto de 2016 (Datasheet MSP430G2553)
NOTA: Para el MSP430G2553: 16kB de flash y 512 bytes de RAM
En el caso del formato PDIP (menor número de pines), no existe el puerto 3.

5.2.1 Pines de Entrada y Salida

Como se puede observar, en la mayoría de pines, la función por defecto es la de pin de Entrada y Salida digitales. Al ser formato PDIP, estos se organizan en dos puertos de 8 pines cada uno, de manera que cada puerto tiene una serie de registros de 8 bits asociados que permiten configurar el funcionamiento de los mismos. De este modo, se pondrá a "0" ó a "1", el bit del pin correspondiente dentro del registro del puerto al que pertenezca, para elegir su función. A modo de resumen, estos son los registros necesarios para cofigurar los pines del puerto 1. Para el puerto 2, simplemente cambiar P1 por P2:

Registro	Nombre código	Explicación	Valor
Direction	P1DIR	Dirección individual de los pines del puerto	0 - Entrada 1 - Salida
Port Select	P1SEL	LSB de vector de 2 bits para selección de función	*
Port Select 2	P1SEL2	MSB del mismo vector	*
Resistor enable	P1REN	Habilita resistencia de pull up/down en entradas digitales	0 - Habilita 1 - Deshabilita
Output	P1OUT	Si pin de salida: Valor de la misma Si pin de entrada: Elegir entre pull-up y pull-down	Si entrada: 0 - Pull-down 1 - Pull-up
Input	P1IN	Valor de la entrada	
Interrupt Enable	P1IE	Habilitar interrupciones	Habilitada si "1"
Interrupt Flag	P1IFG	Bandera para avisar de interrupción pendiente	
Interrupt Edge Select	P1IES	Elegir flanco de subida o de bajada	0 - Flanco Subida 1 - Flanco Bajada

Tabla 5-1. Registros de configuración de pines

En el caso de los registros de Port Select, se elige la función mediante los siguientes valores:

Función del pin	P1SEL2	P1SEL
Entrada/salida digital (Valores por defecto)	0	0
Periférico primario	0	1
Reservado	1	0
Periférico secundario	1	1

Tabla 5-2. Registros Port Select

De esta manera, si queremos configurar un pin, se puede hacer de manera general siguiendo el siguiente ejemplo:

```
P1DIR|=BIT0;
```

Mediante este comando, se hace la función lógica OR de todos los bits del vector P1DIR con un "1" en la posición 0, que es el bit menos significativo. El resultado de esta operación, es el vector P1DIR mantiene todos sus valores iguales, salvo en la posición 0, que se pone a 1. Esto es equivalente a poner el pin 1.0 (pin 0 del puerto 1) como salida. Para ponerlo a 0, (entrada), habría que hacer la AND con el negado de BIT0:

```
P1DIR&=~BIT3; //Pin 1.3 será una entrada
P1REN|=BIT3; //Se habilita la resistencia de pull up o pull down en el pin 1.3
P1OUT|=BIT3; //Como el pin 3 es entrada, sirve para configurar resistencia de pull up
P1OUT|=BIT0; //Como el pin 0 es salida, sirve para ponerla a nivel bajo
```

Para configurar una función diferente:

```
P1SEL|=BIT1+BIT2; //Se configura periférico secundario en pines 1.1 y 1.2 (UART)
P1SEL2|=BIT1+BIT2;
```

Para trabajar con interrupciones:

```
P1IES|=BIT3; //Se configura la interrupción como activa por flanco de bajada
P1IFG=0; //Se borran las flags que haya pendientes
P1IE|=BIT3; //Se activa la interrupción en el pin 1.3
```

Las interrupciones permiten interrumpir la línea principal de ejecución y saltar a una rutina definida como una función a parte. Al acabar la ejecución de la misma, se vuelve al punto de ejecución original y se continúa ejecutando la línea principal. Es necesario habilitar la máscara global de interrupciones (GIE). Se puede hacer con las llamadas:

```
__bis_SR_register(GIE);  ó  __enable_interrupt();
```

Si se usa la primera opción, es posible activar alguno de los modos de bajo consumo que se explicaron anteriormente, de cara a reducir el consumo mientras se espera la activación del flag de una interrupción (xxxIFG):

```
__bis_SR_register(GIE+LPM0_bits);
```

De esta manera, ya se podría completar la configuración de un programa muy básico. Una vez configurados los pines y sus funciones, y activadas las interrupciones si es necesario, lo habitual es entrar en el bucle principal de programa, que se mantiene ejecutándose cíclicamente hasta que salte alguna interrupción si la hubiera. Fuera de la función principal del programa, es necesario definir la rutina de interrupción con una directiva de compilación #pragma, asociándola al vector de interrupción en cuestión (a su dirección). Para ello podemos extrarlas del datasheet del dispositivo (ver tabla 5-3).

Un ejemplo de definición de una rutina de interrupción sería:

```
#pragma vector=PORT1_VECTOR
__interrupt void rutina(void)
{//Codigo rutina}
```

INTERRUPT SOURCE	INTERRUPT FLAG	SYSTEM INTERRUPT	WORD ADDRESS	PRIORITY
Power-Up External Reset Watchdog Timer+ Flash key violation PC out-of-range(1)	PORIFG RSTIFG WDTIFG KEYV(2)	Reset	0FFFEh	31, highest
NMI Oscillator fault Flash memory access violation	NMIIFG OFIFG ACCVIFG(2)(3)	(non)-maskable (non)-maskable (non)-maskable	0FFFCh	30
Timer1_A3	TA1CCR0 CCIFG(4)	maskable	0FFFAh	29
Timer1_A3	TA1CCR2 TA1CCR1 CCIFG, TAIFG(2)(4)	maskable	0FFF8h	28
Comparator_A+	CAIFG(4)	maskable	0FFF6h	27
Watchdog Timer+	WDTIFG	maskable	0FFF4h	26
Timer0_A3	TA0CCR0 CCIFG(4)	maskable	0FFF2h	25
Timer0_A3	TA0CCR2 TA0CCR1 CCIFG, TAIFG (5)(4)	maskable	0FFF0h	24
USCI_A0/USCI_B0 receive USCI_B0 I2C status	UCA0RXIFG, UCB0RXIFG(2)(5)	maskable	0FFEEh	23
USCI_A0/USCI_B0 transmit USCI_B0 I2C receive/transmit	UCA0TXIFG, UCB0TXIFG(2)(6)	maskable	0FFECh	22
ADC10 (MSP430G2x53 only)	ADC10IFG(4)	maskable	0FFEAh	21
			0FFE8h	20
I/O Port P2 (up to eight flags)	P2IFG.0 to P2IFG.7(2)(4)	maskable	0FFE6h	19
I/O Port P1 (up to eight flags)	P1IFG.0 to P1IFG.7(2)(4)	maskable	0FFE4h	18

Tabla 5-3. Vectores de interrupción[12]

El nombre “PORT1_VECTOR” es un define que se puede encontrar la librería “msp430g2553.h”, donde se asocia a la dirección correspondiente. Estos define son:

```
#define PORT1_VECTOR            (2 * 1u)        /* 0xFFE4 Port 1 */
#define PORT2_VECTOR            (3 * 1u)        /* 0xFFE6 Port 2 */
#define ADC10_VECTOR            (5 * 1u)        /* 0xFFEA ADC10 */
#define USCIAB0TX_VECTOR        (6 * 1u)        /* 0xFFEC USCI A0/B0 Transmit */
#define USCIAB0RX_VECTOR        (7 * 1u)        /* 0xFFEE USCI A0/B0 Receive */
#define TIMER0_A1_VECTOR        (8 * 1u)        /* 0xFFF0 Timer0)A CC1, TA0 */
#define TIMER0_A0_VECTOR        (9 * 1u)        /* 0xFFF2 Timer0_A CC0 */
#define WDT_VECTOR              (10 * 1u)       /* 0xFFF4 Watchdog Timer */
#define COMPARATORA_VECTOR      (11 * 1u)       /* 0xFFF6 Comparator A */
#define TIMER1_A1_VECTOR        (12 * 1u)       /* 0xFFF8 Timer1_A CC1-4, TA1 */
#define TIMER1_A0_VECTOR        (13 * 1u)       /* 0xFFFA Timer1_A CC0 */
#define NMI_VECTOR              (14 * 1u)       /* 0xFFFC Non-maskable */
#define RESET_VECTOR            (15 * 1u)       /* 0xFFFE Reset [Highest Priority] */
```

Para poder trabajar con esta librería, es necesario empezar el código incluyendo la librería msp430.h. Normalmente se incluye por defecto al crear un nuevo proyecto en Code Composer Studio. Para poder comprobar el valor al que está asociado una palabra definida en la misma, basta con poner el cursor encima para que lo muestre.

Conociendo todo esto, es posible llevar a cabo la estructura de un programa básico para este microcontrolador. Sería tan solo necesario, revisar las configuraciones específicas de cada uno de los periféricos restantes que aparecen en el diagrama de bloques del microcontrolador (figura 5-2), que están disponibles en el MSP430G2553. Estos periféricos le dotan de un mayor número de funcionalidades y una capacidad de interaccionar y comunicarse con el exterior más amplia y compleja que los pines de entrada y salida vistos anteriormente.

[12] http://www.ti.com/general/docs/lit/getliterature.tsp?genericPartNumber=msp430g2253&fileType=pdf a fecha de Agosto de 2016 (Datasheet MSP430G2553)

5.2.2 Temporizadores

Uno de los periféricos más importantes, sin duda, que están a nuestra disposición, son los timers, mediante los cuales podemos sincronizar tareas, medir intervalos de tiempo. El fundamento de un timer es la de un contador de "ticks" de reloj, de manera que la cuenta se va incrementado o disminuyendo. La capacidad del mismo, viene dado por el número de bits, que determina el valor hasta el cual puede contar. En este caso, poseemos dos timers de 16 bits, lo que permite contar hasta 65 536 valores (0xFFFF). [17]

Otra variable fundamental en el timer, es la frecuencia del reloj con la que se actualiza. En este microcontrolador, poseemos básicamente tres fuentes distintas para el reloj. Por un lado, tenemos el "Auxiliary Clock" o ACLK (por defecto a 12 kHz). Por otro lado, el "Sub-Main Clock" o SMCLK, que suele coincidir con el "Main Clock" o MCLK, que es el usado por la CPU del microcontrolador. El SMCLK es el usado por los periféricos del mismo. Ambos hasta 16 MHz. Por último, se pueden utilizar señales de reloj externa (INCLK o TACLK).

La importancia de la elección del reloj es obvia y dependerá de la aplicación. Contando con el límite de 65 536 valores que imponen los 16 bits de los timers, si el reloj utilizado es varios órdenes de magnitud más lento, el intervalo de tiempo que pasa entre la actualización de la cuenta es mayor. Esto puede ser útil por ejemplo para el parpadeo periódico de un LED, donde no sería necesario una precisión del orden de nanosegundos.

En el caso del MSP430G2553, se tienen dos timer A: Timer0_A3 y Timer1_A3. El número 3 al final se refiere al número de registros "Capture/Compare" (CCR) disponibles que pueden disparar interrupciones de los temporizadores. Éstas serán: TACCR0, TACCR1 y TACCR2. En modo comparación, estos 3 valores nos proporcionan la capacidad de configurar hasta 3 alarmas distintas que activarán la interrupción correspondiente cuando el timer llegue al valor del registro. A la hora de programar en CCS, todos los registros con la forma "TAxxx" y "TA0xxx" se asocian directamente al timer 0. Para poder utilizar el timer 1, es necesario escribir "TA1xxx". [18]

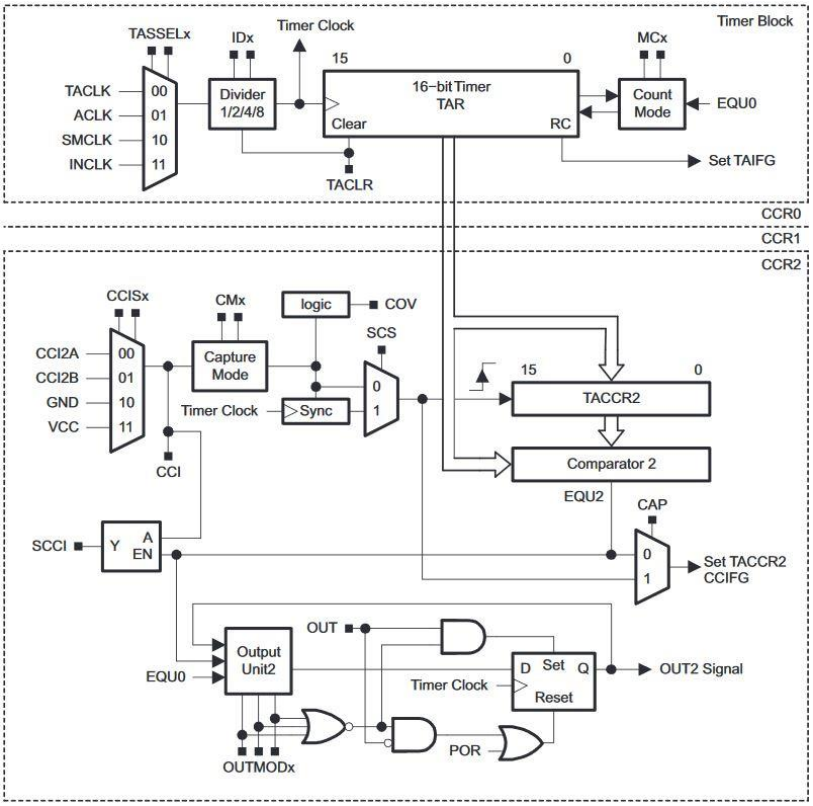

Figura 5-4. Diagrama de bloques de Timer A[13]

[13] http://www.ti.com/lit/ug/slau144j/slau144j.pdf a fecha de Agosto de 2016 (Guía de usuario de la familia MSP430x2xx)

TACCR0, sirve para configurar el periodo del contador, ya que es el valor hasta el que contará. Al alcanzarse se activaría el flag TACCR0-CCIFG. Mediante TACCR1 y TACCR2 se pueden programar alarmas para otros valores del contador, que activarían las banderas TACCR1-CCIFG y TACCR2-CCIFG respectivamente. En todo caso, siempre que se alcance el valor máximo del contador, y pase de nuevo a 0, se activa el flag TAIFG. El valor de cuenta se almacena en el registro TAR.

Dentro del modo comparación, se distinguen tres modos de funcionamiento distintos [19]:

- Modo continuo. El contador empieza a 0 y cuenta hasta 0xFFFF, momento en el cual se activa la interrupción TAIFG. Si se ha especificado un valor en el registro TACCR0 o posteriores, se activa la interrupción correspondiente, pero sigue en todo caso contando hasta 0xFFFF.
- Modo ascendente. El contador comienza en 0 y cuenta hasta el valor almacenado en TACCR0, momento en el que se reinicia y activa la interrupción correspondiente.
- Modo reversible. El contador comienza a 0 y cuenta hasta el valor almacenado en TACCR0. En ese momento, activa la interrupción CCIFG y empieza a contar desde el valor en TACCR0 hasta 0. Cuando llega a 0, se activa la interrupción TAIFG. En este caso, el periodo es el doble que en el modo ascendente.

Por otro lado, el modo "input capture", sirve para contar los pulsos de reloj que transcurren hasta que ocurre determinado evento que se manifiesta como un flanco de subida o de bajada en un pin concreto. Almacena este valor en el registro TACCRx correspondiente.

Por último, existe también la posibilidad de utilizar los timers como "output compare". Mediante este modo de funcionamiento, extensión del modo comparación, se permite la actuación sobre un pin determinado, en función del disparo de interrupciones de los registros TACCRx correspondientes, que se activan con la cuenta de un timer en modo ascendente o reversible. Para configurar la acción a desarrollar, se ha de dar valor a los bits OUTMOD, dentro del registro TACTL, teniendo hasta 7 posibles acciones diferentes. Es muy importante configurar correctamentre en conjunción, el modo del contador y los bits de OUTMOD para obtener la forma deseada a la salida.

OUTMODx	Modo	Explicación
000	Output	Salida controlada por el bit OUT
001	Set	Cambio permanente de 0 a 1 en la salida con TACCRx
010	Toggle/Reset	Toggle en TACCRx. Reset en TACCR0
011	Set/Reset	Set en TACCRx. Reset en TACCR0
100	Toggle	Cambio periódico en la salida con TACCRx. Una vez por ciclo
101	Reset	Cambio permanente de 1 a 0 en la salida con TACCRx
110	Toggle/Set	Toggle en TACCRx. Set en TACCR0
111	Reset/Set	Reset en TACCRx. Set en TACCR0

Tabla 5-4. Modos en Output Compare [19]

Para configurar correctamente el funcionamiento de los timers, es necesario configurar los registros TACTL y TACCTLx.

El registro TACTL, de 16 bits, sirve para configuraciones de los dos TimerA de este microcontrolador. Básicamente, sirve para especificar la fuente del reloj, con los divisores correspondientes de la frecuencia, y el modo de funcionamiento básico. También posee bits destinados al borrado de la cuenta del timer, y a las interrupciones del mismo. Su distribución es la siguiente [19]:

- Bits 15 a 10: Sin definir.
- Bits 9 y 8: TASSELx. Sirve para definir el reloj que se utilizará como fuente (00 TACLK, 01 ACLK, 10 SMCLK, 11 INCLK).
- Bits 7 y 6: ID. Divisores a aplicar a la frecuencia del reloj (00 f/1, 01 f/2, 10 f/4, 11 f/8).
- Bits 5 y 4: MCx. Sirve para definir el modo de funcionamiento (00 parado, 01 ascendente, 10 continuo, 11 reversible).
- Bit 3: Sin definir.
- Bit 2: TACLR. Si se pone a "1", resetea la cuenta del contador (almacenada en el registro TAR).
- Bit 1: TAIE. Habilita interrupción al llegar a 0xFFFF.
- Bit 0: TAIFG. Bandera de interrupción. Se activa al desbordarse el contador.

Para programarlo, simplemente hay que editar los bits con el valor requerido. Se definen en las librerías, palabras reservadas para que sea más intuitivo. Por ejemplo:

```
TACTL = TASSEL_2 + MC_2;
```

Mediante la anterior orden, se configuran los bits del reloj y el modo de funcionamiento de TACTL. Se utiliza TASSEL_2, que es un define de librerías que pone a "10" los bits correspondientes para que el reloj sea el SMCLK. Se hace lo propio para configurar el modo de funcionamiento continuo ("10").

Por otro lado, se tienen tres registros de control TACCTLx de 16 bits por cada timer (TACCTL0, TACCTL1 y TACCTL2), cada uno asociado a los 3 registros CCR correspondientes. Su estructura es la siguiente [19]:

- Bits 15 y 14: CMx. Sirven para definir el modo de "input capture" (00 desactivado, 01 flanco de subida, 10 flanco de bajada, 11 ambos).
- Bits 13 y 12: CCIS. Permite seleccionar el pin de entrada en el que se captura el flanco (00 CCIxA, 01 CCIxB, 10 GND, 11 Vcc).
- Bit 11: SCS. Permite elegir entre captura asíncrona (0) o síncrona (1).
- Bit 10: SCCI. Bit de solo lectura donde se enclava la señal de entrada en el modo capture.
- Bit 9: Sin definir.
- Bit 8: CAP. Su función es seleccionar el modo comparación (0) o captura (1).
- Bit 7, 6 y 5: Bits OUTMODx. Siguiendo los valores de la tabla 5-4, permite seleccionar la actuación sobre el pin de salida en función de la cuenta del timer correspondiente en "output compare" (CAP=0).
- Bit 4: CCIE. Habilita las interrupciones CCIFG de los registros TACCRx.
- Bit 3: CCI. Valor de entrada del pin capture.
- Bit 2: OUT. Valor de salida en el pin sobre el que se actúa en "output compare".
- Bit 1: COV. Permite detectar si se produce una captura nueva antes de que se haya leído correctamente el registroTACCRx correspondiente a la anterior. Se ha de resetear por software.
- Bit 0: CCIFG. Es la bandera correspondiente a la interrupción del registro TACCRx.

La programación de este registro es idéntica a la de TACTL:

```
TACCTL1 = SCS + CM1 + CAP + CCIE;
```

Mediante el ejemplo anterior, se programa el registro CCR1 para actuar en modo captura síncrona con flanco de bajada, y se habilita la interrupción TACCR1-CCIFG.

5.2.3 Módulos de comunicación

Otra parte de vital importancia del MSP430G2553 son los periféricos destinados a establecer protocolos de comunicación con el exterior. Se disponen básicamente de dos grandes módulos: USCI_A y USCI_B. Ambos implementan comunicaciones serie y son configurables para adaptarse a distintos protocolos. En realidad, USCI_A y USCI_B son dos canales independientes de una misma USCI, de manera que, en dispositivos más avanzados, existe un mayor número de USCI con la notación: USCI_A0, USCI_B0, USCI_A1…etc.

Por un lado, el módulo USCI_A, implementa comunicaciones síncronas (SPI) y asíncronas (UART, IrDA y LIN). Por otro lado, el módulo USCI_B implementa exclusivamente comunicaciones síncronas (SPI, I^2C). Cada uno puede sólo implementar un protocolo al mismo tiempo, pero USCI_A y USCI_B pueden trabajar simultáneamente, por lo que se pueden mantener dos comunicaciones distintas al mismo tiempo. Todos los registros de configuración relativos a los módulos de comunicación, tienen la forma UCAxxx (para el USCI_A) o UCBxxx (para el USCI_B). A continuación, se detallará el funcionamiento y la programación de los protocolos UART y SPI, que son los que se utilizarán en el proyecto:

El protocolo UART, ya abordado de manera resumida en el segundo apartado relativo al protocolo MIDI, se trata de un protocolo de comunicación serie asíncrona (sin señal de reloj). Es importante que los dos dispositivos conectados mediante una UART "pacten" de antemano la velocidad de comunicación, y que la información viaje en tramas bien definidas (normalmente un único byte más algunos bits adicionales de comienzo, parada o paridad) para que los dispositivos puedan mantenerse sincronizados. La estructura típica de un mensaje enviado por UART es la explicada en el apartado 2.2, por lo que no se volverá a insistir en la misma.

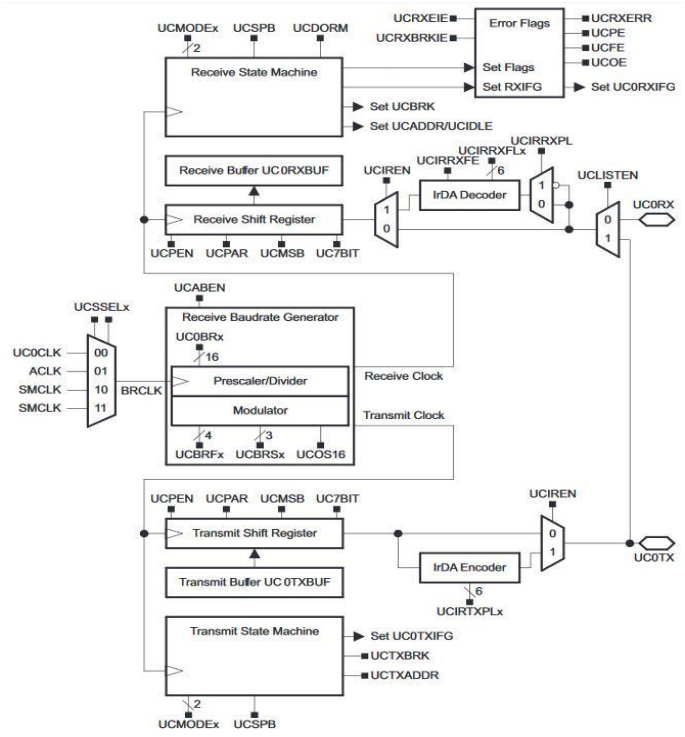

Figura 5-5. Diagrama de bloques de USCI_A en modo UART [19]

En el MSP430G2553, existe tan solo la opción de configurar una única UART, en el USCI_A. En otros microcontroladores superiores, se puede encontrar un mayor número de las mismas. Para poder utilizarla, es primeramente necesario configurar los pines 1.1 y 1.2 mediante los registros P1SEL en su funcionalidad secundaria, que corresponde, respectivamente, a la línea RXD (recepción de datos) y TXD (transmisión). La operación de la misma, suele ser habitualmente esperar la interrupción correspondiente para la llegada de datos por la línea RXD, y mandar la información necesaria por la línea TXD tras comprobar que esté libre, para no perder ningún dato que se quiera enviar. Los caracteres de entrada se cargan al registro UCA0RXBUF, activando la bandera UCA0RXIFG cuando un byte está disponible. Para el envío de un dato, se escribe en el registro UCA0TXBUF. Es necesario que la bandera UCA0TXIFG esté a nivel alto, indicando que el registro de salida está libre. Las banderas UCA0RXIFG y UCA0TXIFG se resetean automáticamente cuando acaba la operación de lectura o escritura.

La configuración de los diferentes parámetros se lleva a cabo de forma similar a los timers. En este caso, son los registros de control de 8 bits UCA0CTL0 y UCA0CTL1 los registros clave para determinar el funcionamiento de la comunicación UART en el dispositivo. Estos se utilizan también para el SPI de la USCI_A. El primer 0 hace referencia a la USCI_A0, aunque en este dispositivo es la única disponible.

La estructura de UCA0CTL0 es la siguiente [19]:

- Bit 7: UCPEN. Si se pone a nivel alto, habilita el bit de paridad para la comunicación.
- Bit 6: UCPAR. Sólo tiene utilidad si UCPEN=1. Configura la paridad de la comunicación como impar (0) o par (1).
- Bit 5: UCMSB. Especifica si la transmisión empieza por el bit LSB (configurar a 0), o MSB (configurar a 1). Por defecto, el estándar RS-232 empieza por el LSB, por lo que el registro está por defecto a 0. El protocolo MIDI también comienza por el LSB.
- Bit 4: UC7BIT. Elige entre transmisión de palabras de 8 bits (valor por defecto, a 0), o de 7 bits (valor a 1). Éste último caso puede ser útil para la transmisión de caracteres ASCII (7 bits).
- Bit 3: UCSPB. Configura el número de bits de stop en la comunicación. A nivel bajo, 1 bit de stop. Si se pone a nivel alto, 2 bits de stop.
- Bits 2 y 1: UCMODE. Configura el modo de funcionamiento del puerto si la comunicación es síncrona (00 SPI de 3 pines, 01 SPI de 4 pines con UCxSTE activo a 1, 10 SPI de 4 pines con UCxSTE activo a 0, 11 I^2C) o asíncrona (00 UART con funcionamiento normal, 01 ó 10 Modo "multiprocessor" con autodetección de la dirección cuando más de dos dispositivos comparten el mismo bus, 11 UART con detección automática del baudrate).
- Bit 0: UCSYNC. Permite elegir entre comunicación asíncrona (0) o síncrona (1).

La estructura de UCA0CTL1 es la siguiente [19]:

- Bits 7 y 6: UCSSEL. Elige el reloj para definir el baudrate (00 UCLK, 01 ACLK, 10/11 SMCLK).
- Bit 5: UCRXEIE. Si se pone a nivel alto, habilita interrupción de recepción de caracteres erróneos.
- Bit 4: UCBRKIE. Habilita la interrupción, al configurarse a nivel alto, de la recepción de "serial breaks", que suceden cuando la línea de recepción se mantiene a nivel bajo durante un tiempo mayor que el tiempo de trama.
- Bit 3: UCDORM. Sirve para introducir el módulo USCI en "sleep mode".
- Bit 2: UCTXADDR. Bit de gran utilidad para trabajar con la UART en modo "multiprocessor". Si vale "0", la siguiente trama será un dato. Si se pone a "1", será una dirección.
- Bit 1: UCTXBRK. Al ponerse a nivel alto, prepara la UART para transmitir un "serial break".
- Bit 0: UCSWRST. Resetea la UART por software al estar a nivel alto. Es siempre necesario, para cambiar la configuración de la UART, llevar a cabo primero un reset, y una vez se han hecho las configuraciones oportunas, poner el bit a nivel bajo.

Para habilitar las ya comentadas interrupciones, es necesario poner a nivel alto los bits UCA0TXIE y UCA0RXIE en el registro de habilitación de interrupciones IE2. Las banderas UCA0TXIFG y UCA0RXIFG son los bits LSB del registro IFG2.

Otro registro que puede ser de utilidad para la operación de la UART, es el registro de 8 bits UCA0STAT, que contiene información sobre el funcionamiento de la misma y errores que se puedan ir detectando en la comunicación. Su estructura es la siguiente [19]:

- Bit 7: UCLISTEN. Permite activar, si se configura a 1, el dispositivo en modo "loopback", de manera que lo transmitido por TXD se realimenta a la entrada RXD. Puede ser útil para comprobar el funcionamiento correcto de la UART.
- Bit 6: UCFE. Detección de errores en la trama. Por ejemplo, un bit de stop a nivel bajo.
- Bit 5: UCOE: Detección de "overruns". Tal situación ocurre cuando llega un nuevo carácter al buffer de lectura sin haber leído el anterior. Se pone a nivel bajo automáticamente cuando el carácter es leído. No se debe de borrar por software.
- Bit 4: UCPE. Detección de errores en la paridad. Sucede cuando el número de "1s" en la transmisión no encaja con el bit de paridad.
- Bit 3: UCBRK. Detección de un "serial break".
- Bit 2: UCRXERR. Bandera de detección de errores. Estará acompoáda de UCFE, UCOE o UCPE.
- Bit 1: UCADDR. Se pone a nivel alto cuando una dirección es recibida correctamente en modo "multiprocessor".
- Bit 0: UCBUSY. El bit se mantiene a nivel bajo si no se está realizando ninguna operación. Mientras una operación de escritura o lectura se lleva a cabo, se pone a nivel alto.

Otro factor esencial a la hora de configurar la UART, es la configuración del baudrate. Para ello, es fundamental elegir el reloj adecuado y los divisores de frecuencia requeridos para obtener el baudrate que se necesita. Para elegir el reloj, basta con configurar los bits UCSSEL en el registro UCA0CTL1. Para escalar la frecuencia del reloj al baudrate requerido, se disponen una serie de registros de 8 bits: UCA0BR0, UCA0BR1 y UCA0MCTL. El funcionamiento de estos es explicado de manera detallada en la guía de usuario que ofrece Texas Instruments para la familia MSP430x2xx. Básicamente, para alcanzar el baudrate requerido, es necesario dividir la frecuencia del reloj entre el baudrate, de manera que el número resultante será por lo general un número decimal. Este número, tendrá que ser codificado mediante los registros anteriormente mencionados. La parte entera, en hexadecimal, se guarda en UCA0BR0 y UCA0BR1, escribiendo la parte más significativa en UCA0BR1 y la menos significativa en UCA0BR0. De manera que tenemos hasta 16 bits para codificar el resultado de la división. Esto es equivalente a coger este resultado, hacer la división entera por 256, y almacenar el entero resultante en UCA0BR1. El resto de la división hasta 256, se almacena en UCA0BR0. Por último, la parte decimal se aproxima mediante el registro de modulación UCA0MCTL. Por ejemplo, para la velocidad MIDI estándar de 31 250 bps con el reloj de 16 MHz:

$$\frac{16\ MHz}{31\ 250} = 512;\ \ 512 = 0x0200\ (hex)$$

De esta forma en el registro UCA0BR1 se almacenaría un 0x02, mientras que en el UCA0BR0 se almacena 0x00. Como no hay parte decimal, UCA0MCTL será 0x00 también.

Para mayor detalle, consultar la guía de usuario de la familia de microcontroladores MSP430x2xx aportada por el fabricante Texas Instruments [19], donde se ven estas ecuaciones con mayor profundidad y se aportan tablas para conseguir los baudrates más estandarizados mediante diversas frecuencias de reloj. También existen herramientas online que permiten calcular los registros directamente[14].

[14] http://www.daycounter.com/Calculators/MSP430-Uart-Calculator.phtml a fecha de Agosto de 2016.

Tanto la USCI_A como la USCI_B, tienen la posibilidad de ser configuradas para comunicarse mediante protocolo SPI con otros dispositivos. Éste se basa en un protocolo master-salve donde el dispositivo maestro decide cuando comienza el intercambio de datos y con qué dispositivo esclavo se produce. Para la USCI_A, es necesario poner el bit UCSYNC a 1, mientras que en la USCI_B, que solo tiene comunicaciones síncronas, es necesario configurarlo mediante los bits CMODE, y elegir el modo de funcionamiento requerido. En este caso, tenemos la opción de funcionar con un SPI de 3 líneas:

- UCxSOMI (MISO): Línea de Slave Out – Master In. Depende de si el microcontrolador se configura como "master" o "slave", la línea será entrada o salida de datos respectivamente.
- UCxSIMO (MOSI): Línea de Slave In – Master Out. Depende de si el microcontrolador se configura como "master" o "slave", la línea será salida o entrada de datos respectivamente.
- UCxCLK: Línea de reloj, siempre aportado por el "master" de la comunicación.

También puede funcionar en modo de 4 líneas, añadiendo a las tres anteriores:

- UCxSTE: Habilita transmisión a esclavo cuando un mismo bus es compartido por varios maestros, evitando el conflicto entre estos.

Tenemos dos registros, uno para TXD y otro para RXD, de manera que podemos llevar a cabo una comunicación "full-duplex". En general, es necesario utilizar un pin de entrada y salida como "Chip Select" en modo de 3 líneas, modificándolo por software para comunicarse con el dispositivo esclavo requerido.

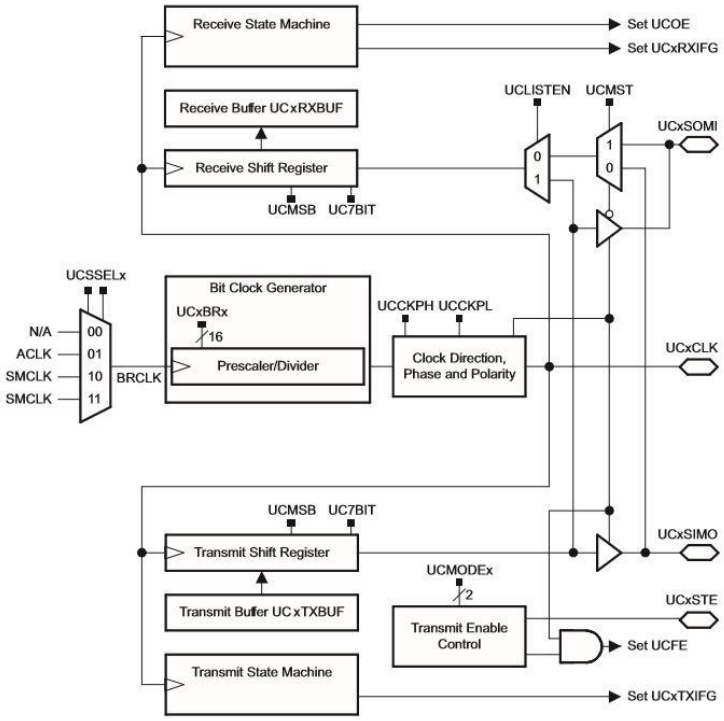

Figura 5-6. Diagrama de bloques de USCI en modo SPI[15]

La configuración y el funcionamiento del SPI es similar al explicado anteriormente para trabajar con una UART. Es, en primer lugar, necesario llevar a cabo las configuraciones con el reset activo, para finalmente desactivar el reset. Posteriormente, se han de activar las interrupciones UCxRXIE y UCxTXIE para la llegada y la transmisión de datos. Éstas se encuentran en el registro IE2. Como en el caso de la UART, estos pueden configurarse como palabras de 8 ó 7 bits y se reciben en el registro UCxRXBUF y se han de escribir en el registro UCxTXBUF.

[15] http://www.ti.com/lit/ug/slau144j/slau144j.pdf a fecha de Agosto de 2016 (Guía de usuario de la familia MSP430x2xx)

De cara a la configuración, volvemos a tener los mismos registros vistos para la UART, sólo que en este caso hay que seleccionar UCSYNC a 1 y el modo correspondiente. Una vez hecho esto, el significado de los distintos bits del registro varía consecuentemente, para configurar los parámetros del SPI. En el caso del registro UCx0CTL0 (sería UCA0CTL0 para la USCI_A y UCB0CTL0 para la USCI_B), tendríamos la siguiente estructura [19]:

- Bit 7: UCCKPH. Permite configurar la fase del reloj. A nivel bajo, el dato se sobreescribe en el primer flanco del reloj, y el nuevo dato se captura en el siguiente. A nivel alto, el dato es primeramente capturado en el primer flanco y se sobreescribe en el segundo.
- Bit 6: UCCKPL. Sirve para configurar la polaridad del reloj. Si vale 0, se considera el estado inactivo cuando el reloj está a nivel bajo. Si vale 1, cuando está a nivel alto.
- Bit 5: UCMSB. Especifica si la transmisión empieza por el bit LSB (configurar a 0), o MSB (configurar a 1).
- Bit 4: UC7BIT. Elige entre transmisión de palabras de 8 bits (valor por defecto, a 0), o de 7 bits (1).
- Bit 3: UCMST. Permite elegir si el microcontrolador actuará como maestro o esclavo.
- Bits 2 y 1: UCMODE. Configura el modo de funcionamiento del puerto (00 SPI de 3 pines, 01 SPI de 4 pines con UCxSTE activo a 1, 10 SPI de 4 pines con UCxSTE activo a 0, 11 I^2C).
- Bit 0: UCSYNC. Permite elegir entre comunicación asíncrona (0) o síncrona (1).

La estructura de UCx0CTL1 es la siguiente [19]:

- Bits 7 y 6: UCSSEL. Elige el reloj para definir el baudrate (00 N/A, 01 ACLK, 10/11 SMCLK). Si el dispositivo es esclavo, se utiliza la línea UCxCLK como entrada con la fuente de reloj.
- Bits 5 a 1: Sin definir.
- Bit 0: UCSWRST. Reset. Necesario previo a configuración

En el caso del registro UCx0STAT [19]:

- Bit 7: UCLISTEN. Permite activar, si se configura a 1, el dispositivo en modo “loopback”, de manera que lo transmitido por TXD se realimenta a la entrada RXD.
- Bit 6: UCFE. Detección de errores en la trama. No se utiliza en modo 3 líneas o en modo esclavo. Tan sólo para indicar algún conflicto en el modo de 4 líneas.
- Bit 5: UCOE: Detección de “overruns”. Tal situación ocurre cuando llega un nuevo carácter al buffer de lectura sin haber leído el anterior. Se pone a nivel bajo automáticamente cuando el carácter es leído. No se debe de borrar por software.
- Bits 4 a 1: Sin definir.
- Bit 0: UCBUSY. El bit se mantiene a nivel bajo si no se está realizando ninguna operación. Mientras una operación de escritura o lectura se lleva a cabo, se pone a nivel alto.

Como en el caso de la UART, es necesario configurar la velocidad de la transmisión. En este caso es más sencillo porque se prescinde del registro UCA0MCTL (que tendrá que mantenerse a 0 para un funcionamiento correcto). Para escalar el reloj, se utilizan de nuevo los registros UCx0BR0 y UCx0BR1, del mismo modo explicado anteriormente para la UART.

Por último, se consta también de la posibilidad de programar la USCI_B en modo comunicación I^2C. Este protocolo no será utilizado en el proyecto, por lo que no se abordará con profundidad. El funcionamiento es similar al del SPI. En este caso, son tan sólo necesarios dos líneas, una línea de reloj (SCL) y otra de datos bidireccional (SDA) que cambia siempre con el reloj a nivel bajo, salvo en el comienzo y final de la comunicación. Es necesario enviar primero la dirección del esclavo y recibir la confirmación de que está disponible antes de enviar el dato. La configuración y el funcionamiento son similares a los explicados anteriormente y se utilizan los mismos registros. Se puede consultar la guía de usuario para mayor detalle.

5.2.4 Otros periféricos

En este apartado se describirán brevemente el resto de periféricos presentes en el MSP430G2553:

El módulo "Brownout Protection", funciona de manera transparente al usuario. Se trata de un circuito destinado a la protección de dispositivos ante caídas en la alimentación, permitiendo una distribución adecuada de la señal de reset del sistema en el encendido y el apagado.

El módulo "Watchdog", activado por defecto, tiene como función actuar frente a un cuelgue o error del sistema, provocando un reseteo controlado del mismo al desbordar el contador asociado al mismo. Si no es necesario, se puede desactivar al comienzo del programa:

```
WDTCTL = WDTPW + WDTHOLD;
```

Para permitir la lectura de sensores analógicos, el MSP430G2553 cuenta también con un ADC de 10 bits de resolución y de 200 ksps de velocidad de conversión (el periodo de sampleo es programable). Viene con un sensor de temperatura integrado. Se pueden seleccionar referencias de tensión internas (1.5V ó 2.5V) y externas. Como señal de reloj, a parte de las fuentes habituales, se puede escoger un oscilador interno del ADC que oscila en torno a los 5 MHz.

La inicialización de una conversión se puede comenzar mediante software o mediante una interrupción del timer. Cuando el valor de tensión a la entrada es convertido a su valor digital de 10 bits correspondiente, se guarda en el registro de 16 bits ADC10MEM. Los 6 bits MSB del registro se quedan a 0 en todo momento, y se utilizan los bits restantes para guardar el valor. Es necesario elegir la entrada a convertir entre las 8 posibles entradas multiplexadas (A0, A1, A2 y sucesivos) para una única conversión, o la mayor de ellas para una sucesión de conversiones. Para facilitar conversiones seguidas, se incorpora un bloque de transferencia automática (DTC), que transporta el valor que se almacena en ADC10MEM, directamente a otro registro en memoria sin necesidad de controlar esta transferencia por software.

Como en los otros periféricos se disponen una serie de registros de control para configurar el funcionamiento del ADC. En este caso, poseemos los registros ADC10CTL0 y ADC10CTL1 para el funcionamiento general, y los registros ADC10DTC0 y ADC10DTC1 para configurar el DTC. Los parámetos configurables más importantes son la elección de reloj, la velocidad de sampleo, las referencias de voltaje, las entradas, las interrupciones y los modos de funcionamiento. Éste último, tiene una especial relevancia y se configura con los bits CONSEQx. Básicamente, permite configurar el ADC en modo de conversión única (00), conversión única en diversos canales (01), sucesión de conversiones en un único canal (10), o sucesión de conversiones en diversos canales (11). En cuanto a las interrupciones, tiene asociadas el flag ADC10IFG y el registro de habilitación ADC10IE.

Por último, se posee también un módulo comparador analógico, conocido en esta familia de microcontroladores como "Comparator_A+". Su función es comparar el voltaje de los dos terminales a la entrada. Si el terminal "positivo" es mayor que el "negativo", la salida del comparador (CAOUT) es un "1" lógico. De lo contrario, la salida se mantiene a nivel bajo. Los terminales de entrada se pueden elegir de entre los varios pines disponibles que se encuentran multiplexados a la entrada del comparador (CA0, CA1, CA2 y sucesivos). La salida del comparador puede alimentar la entrada del timer A en modo captura. Ésta tiene asociada la bandera de interrupción CAIFG que se pone a nivel alto cuando se tiene un cambio en el valor de la salida. La interrupción se habilita mediante el bit CAIE. Se posee un filtro RC a la salida para evitar oscilaciones a la salida si la diferencia de los voltajes a la entrada es pequeña. Se puede habilitar mediante software.

De manera similar a los periféricos anteriormente mencionados, posee sus propios registros de configuración CACTL1 y CACTL2, que permiten definir diferentes parámetros, como pueden ser la activación de la interrupción, del filtro RC, selección de entrada o selección de referencias, entre otros.

6 PROGRAMACIÓN MICROCONTROLADORES

En los capítulos anteriores, se ha descrito tanto el protocolo MIDI, como los distintos dispositivos que se utilizarán para desarrollar el proyecto. A continuación, se procederá a describir la solución a nivel de software implementada para llevar a cabo el controlador MIDI. En particular, se verá el código realizado tanto para el microcontrolador que se encarga de gestionar el receptor, como el encargado de la gestión del bloque emisor. Para ello, se utilizarán conceptos y utilidades repasadas en los capítulos previos, por lo que se evitará hacer un repaso exhaustivo de los mismos de nuevo. La solución implementada se verá en este capítulo de manera esquemática, para mayor detalle se pueden consultar los anexos adjuntados al final.

6.1 Sistema de desarrollo y herramienta software

Como ya se ha comentado en el anterior capítulo, el fabricante Texas Instruments ofrece para facilitar el desarrollo de aplicaciones para sus microcontroladores una serie de placas o "Launchpads" que permiten conectar los microcontroladores y poder programarlos de manera sencilla a través del puerto serie que implementan bajo conexión USB, así como acceder a la mayoría de sus funcionalidades mediante pines. Para el microcontrolador MSP430G2553, Texas Instruments ofrece el Launchpad MSP-EXP430G2. Éste incluye entre otros, dos botones asociados al pin 1.3 y al reset, y dos leds asociados a los pines 1.0 y 1.6; así como diversos jumpers para cambiar algunas configuraciones.

La placa Launchpad, se conecta a un PC mediante un cable USB, y automáticamente se deben de instalar los drivers necesarios para poder comunicarse con la placa mediante el puerto serie. Para llevar a cabo la programación del microcontrolador, en lenguaje C, Texas Instruments ofrece el programa "Code Composer Studio", una herramienta software de desarrollo de aplicaciones basada en Eclipse, que permite crear proyectos asociados a los distintos microcontroladores de Texas Instruments, desarrollar el código correspondiente basándose en las librerías que aporta el fabricante para sus microcontroladores, y compilar y debuggear el mismo. Adicionalmente, ofrece otra serie de servicios, como el Grace, para la edición de los distintos parámetros y registros del microcontrolador de forma gráfica, o el repositorio de ejemplos MSP430ware, útil para conocer la programación básica de los distintos periféricos.

Para trabajar con CCS, se creará primero el proyecto mediante la opción: *Project -> New CCS Project*. Se abre una ventana que permite configurar ciertos aspectos básicos del proyecto, como el nombre, la ubicación, y el microcontrolador para el que se realizará. Es importante elegir correctamente el puerto en el que se encuentra conectado el Launchpad para poder transferir el código posteriormente.

A partir de aquí, el procedimiento es bastante parecido a cualquier compilador de C. En el nuevo proyecto se ha incluido por defecto la librería <msp430.h>, que incluye por defecto las definiciones básicas para el funcionamiento del microcontrolador y poder trabajar con sus registros de manera sencilla. Se pueden añadir más librerías al proyecto, y nuevos archivos .c, siempre teniendo en cuenta el límite de capacidad del microcontrolador donde se escribirá el programa.

Una vez finalizado el código, es conveniente repasar mediante la opción "*Build"*, que no existan errores en el mismo. Mediante la opción "*Debug*", se puede escribir el programa en el dispositivo para comprobar su funcionamiento más allá de posibles errores de sintáxis que se pudieran encontrar en "*Build*".

Escrito el programa en el dispositivo, se puede proceder a su ejecución, pausar la misma, borrarlo, y utilizar otra serie de utilidades que se ofrecen para llevar a cabo la depuración del código, como la adición de Breakpoints y la posibilidad de seguir la evolución de las variables que se configuren como tal.

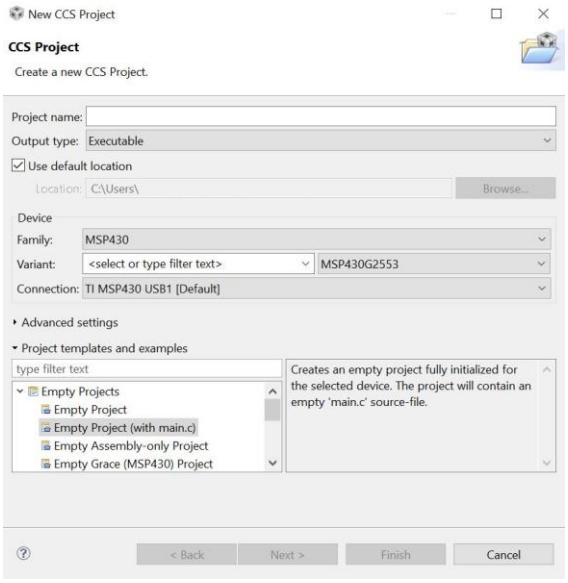

Figura 6-1. Ventana de creación de un proyecto

6.2 Programación microcontrolador receptor

Una vez se ha repasado el funcionamiento del software CCS y de la placa de desarrollo que ofrece el fabricante Texas Instruments, se procederá a describir la solución adoptada a nivel de software para implementar el bloque receptor, que se encarga de recibir comandos MIDI por Bluetooth y transmitirlos a un dispositivo MIDI.

Para ello, es necesario configurar la USCI_A para trabajar como una UART a 115 200 bps, velocidad a la que se configuraron los módulos Bluetooth. Para comunicarse con el instrumento MIDI, sería necesario otra UART que trabajase a 31 250 bps, velocidad estándar de MIDI. Sin embargo, la otra USCI que incorpora el microcontrolador solo posee modos de comunicación síncrona. Por este motivo, será necesario programar uno de los timers existentes para trabajar como una software UART, gracias a los modos "output compare" que se explicaron en el capítulo anterior. Para esto, se puede seguir la guía de aplicación "*Implementing a UART Function With TimerA3*" [20] de Texas Instruments, así como diversos ejemplos que existen en la herramienta MSP430ware útiles para este caso, como el ejemplo: "*Timer_A, Ultra-Low Pwr Full-duplex UART 9600, 32kHz ACLK*". El hecho de utilizar las características hardware de un timer del MSP430, permite que la demanda de recursos de CPU de la UART software no sea mucho mayor respecto de la UART hardware, además de poder proporcionar una temporización adecuada para la comunicación.

La primera decisión que se debe de tomar es una correcta distribución de los pines que permitan distribuir los distintos periféricos que se utilizarán en pines libres. Como la UART hardware (conexión MSP430-módulo HC05), solo puede programarse en la USCI_A, su localización será inmediata. Los pines 1.1 y 1.2 se tendrán que configurar en su funcionalidad secundaria, que corresponde a las líneas de RXD y TXD de una UART, respectivamente. Posteriormente, se ha de abordar la programación del periférico para llevar a cabo una comunicación UART, como se vio en el anterior capítulo.

Como reloj, se utilizará el SMCLK, que se ha configurado previamente a 8 MHz mediante el DCO. Esto se tendrá que tener en cuenta para la configuración del baudrate. Para la elección de la velocidad influyó el incorrecto funcionamiento de la UART software a velocidades menores. Mediante un osciloscopio, se pudo comprobar que las interrupciones iban acumulándose sin tener tiempo suficiente a ejecutarse completamente. Además, la velocidad debía de ser múltiplo del baudrate de MIDI, para poder cargar un valor entero al registro correspondiente. Los registros más relevantes para configurar la UART, son UCA0CTL0 y UCA0CTL1. El primer registro no es necesario modificarlo, ya que por defecto están todos los bits a 0, ofreciendo la configuración adecuada para una comunicación MIDI: dato de 8 bits con un bit de stop y sin bit de paridad, comenzando por el bit LSB. Funcionamiento asíncrono en modo UART normal.

En el caso del registro UCA0CTL1, es necesario modificar los bits UCSSEL para elegir el SMCLK como fuente de reloj (la que se configuró a 8 MHz). Para hacer esto, es primero necesario poner el bit de reset UCSWRST de este mismo registro a nivel alto. Mientras el reset se mantenga en este estado, es posible llevar cabo todas las configuraciones necesarias. Una vez éstas sean finalizadas, se puede desactivar el reset. En este caso, es necesario tan sólo, darles el valor a los bits UCSSEL de "10". Esto es equivalente a escribir, utilizando defines de la librería:

```
UCA0CTL1 |= UCSSEL_2; // Elegimos reloj SMCLK. UCSSEL es "10" ya que UCSSEL_2 es 0x80
```

Posteriormente, es necesario llevar cabo la configuración del baudrate. Para ello se utilizan los registros UCA0BR0, UCA0BR1 y UCA0MCTL. Si dividimos 8 MHz entre el baudrate (115 200 bps), el resultado de la división es 69.444… La parte entera, no llega a 256, por lo que se guarda en el registro UCA0BR0. La parte decimal, se codifica en el registro UCA0MCTL.

Por último, es necesario habilitar en el registro IE2 la interrupción para la llegada de datos UCA0RXIE.

De esta manera, se ha finalizado la configuración de la UART hardware que se encarga de comunicarse con el módulo HC-05. Tan sólo resta configurar un timer A para controlar un pin de salida como si fuera una línea TXD UART. Para ello se utiliza el modo "output compare", que permite ir cambiado alguno de los registros CCR, e ir actuando en función de la cuenta y de los bits OUTMODx sobre el pin de salida. Elegiremos el Timer1, en los pines 2.0 y 2.3, que se han de configurar en su funcionalidad primaria. Se pone el pin 2.0 como salida (será el TXD) y el 2.3 como entrada, que sería la línea RXD, aunque en este caso no se utilizará, ya que la lectura de datos se realiza por la UART hardware. En caso de querer implementar la línea RXD de una UART por software, es necesario acudir a otro registro CCR y configurarlo en modo captura síncrona.

A continuación, se procede a la configuración de los registros del timer. En el registro TA1CTL, es necesario elegir el reloj SMCLK (TASSEL_2) y seleccionar el modo continuo (MC_2). En el caso del registro TA1CCTL0, asociado al CCR0, es tan sólo necesario poner a 1 el bit OUT, de manera que la salida del timer esté a 1 por defecto (línea a nivel alto en reposo). Por último, se configura el registro asociado al CCR1, el TA1CCTL1, para funcionar en modo de captura síncrona en flanco de bajada y se habilita la interrupción del registro CCR1. Este paso es útil para utilizar la UART software como RXD, aunque no se utlizará en este caso. La interrupción del registro CCR0, que es utilizado como TXD, permanece desactivada hasta que comience una transmisión.

Realizadas todas las configuraciones necesarias, se ha de activar la máscara global de interrupciones, previamente a entrar en un bucle while(1) donde se espera a la llegada de algún dato por la UART hardware. Para ello es necesario programar una rutina de interrupción como se vio en el capítulo anterior que se ejecute a la llegada de un dato al buffer de entrada UCA0RXBUF. En la rutina de interrupción, se guarda este dato en una variable en memoria de tipo "unsigned char", y se pone a 1 un entero auxiliar que se utiliza a modo de bandera. Finalizada la rutina de interrupción, se vuelve al bucle donde se pregunta por el estado de la misma, y si está a nivel alto, se llama a la función "TimerA_UART_tx", con el dato de argumento, y se baja la bandera.

La misión de esta función es enviar el dato mediante la UART software. Para ello, tras asegurar que la última transmisión finalizó, permaneciendo en un bucle while hasta que la interrupción de CCR0 esté desactivada (sólo sa activa durante una transmisión), se actualiza el registro CCR0 con el valor del bit time, que es el tiempo que ha de perdurar un bit en un estado determinado antes de pasar al siguiente (ha de ser un valor entero). Es producto de la división de la frecuencia de reloj (8 MHz) entre el baud rate (31 250 bps). Una vez cargado el bit time en el registro TA1CCR0, es necesario activar la interrupción CCIFG de este registro en TA1CCTL0. Adicionalmente, es necesario, para asegurarse que la línea está a 1 (reposo), activar el modo OUTMOD0, que es equivalente a poner los bits OUTMODx en "001". Esto ejecuta un set en la salida del timer, poniéndolo a 1 de manera indefinida (hasta que los bits OUTMOD sean modificados nuevamente con distinto valor). Por último, y para facilitar el trabajo de la interrupción asociada al Timer que controla la acción sobre el pin de salida, es conveniente construir una nueva variable, llamada "txData", que incluya el dato de 8 bits que constituye el mensaje MIDI que ha llegado, y los bits de start y de stop. Para ello es necesario hacer una OR entre el byte y el valor 0x100 para poner un 1 como MSB en el noveno bit. Éste será el bit de stop, que siempre ha de valer 1 y debe de ser el último porque es una transmisión que empieza en el LSB. Para añadir el bit de start en la posición 0, basta con trasladar todos los bits hacia la izquierda con la operación "<<", creando un 0 en la primera posición.

Una vez actualizado el registro CCR0 y se ha preparado el mensaje para ser transmitido en la variable "txData", la rutina de interrupción del timer saltará cuando se complete el primer bit time. En ésta, se vuelve a cargar de nuevo el tiempo de bit para mantener el nuevo bit que se va a transmitir por la línea el tiempo necesario. Posteriormente, se ha de comprobar si quedan bits en "txData" por enviar o ya se han transmitido todos los bits del dato. Para ello, se define una variable auxiliar "txBitCnt", inicializada a 10, y que representa el número de bits restantes por transmitir. A este valor se le va restando una unidad según se transmiten los bits. En caso de que haya llegado a 0 y la interrupción salte, se resetea la cuenta a 10 y se desactiva la interrupción. La línea permanece con el último valor que tuviera, que siempre será a nivel alto, ya que toda transmisión acaba con un bit de stop (1 lógico). Esto prepara la línea de transmisión para permanecer en reposo hasta que un nuevo dato haya llegado. Con la llegada de este nuevo dato, se vuelve a llamar a la función "TimerA_UART_tx" que activa de nuevo la interrupción para que el timer vuelva a actuar.

En el caso de que la cuenta no sea 0 cuando salta la interrupción, es necesario mandar el bit correspondiente. Para ello, se envía el bit que en ese momento sea LSB en "txData". Al comienzo de la transmisión, éste será siempre el bit de start (0 lógico), pero según se envían los bits, se actualiza la variable con la operación ">>" para mover todos los bits a la derecha. El envío del bit correspondiente, se hace jugando con el valor de los bits OUTMODx, que al comienzo de toda transmisión se pone en OUTMOD0, para hacer un set en la línea. En el caso de que el bit a transmitir valga 0, se activa OUTMOD2, que pone el bit 7 del registro TA1CCTL0 a nivel alto. Al hacerlo con una OR, el valor del bit 5 no varía, por lo que el valor de los bits OUTMODx es de "101", que corresponde a hacer un reset en la línea permanente hasta un nuevo cambio en estos bits. Si el próximo bit es un 1 lógico, basta con hacer la AND con el negado de OUTMOD2 para que el bit 7 valga de nuevo 0, y por lo tanto los bits OUTMODx valgan "001" que se corresponde de nuevo con un set a la línea. Efectuadas estas operaciones, se resta una unidad a la cuenta de bits y se desplaza la variable "txData" hacia la derecha". La interrupción se desactiva al mandar todos los bits del dato, y por lo tanto la línea queda en reposo hasta que se necesite enviar un nuevo mensaje.

A continuación, se adjunta un diagrama de flujo para aclarar el funcionamiento del programa. Para mayor detalle, consultar el anexo I donde se puede encontrar el código completo para el bloque receptor.

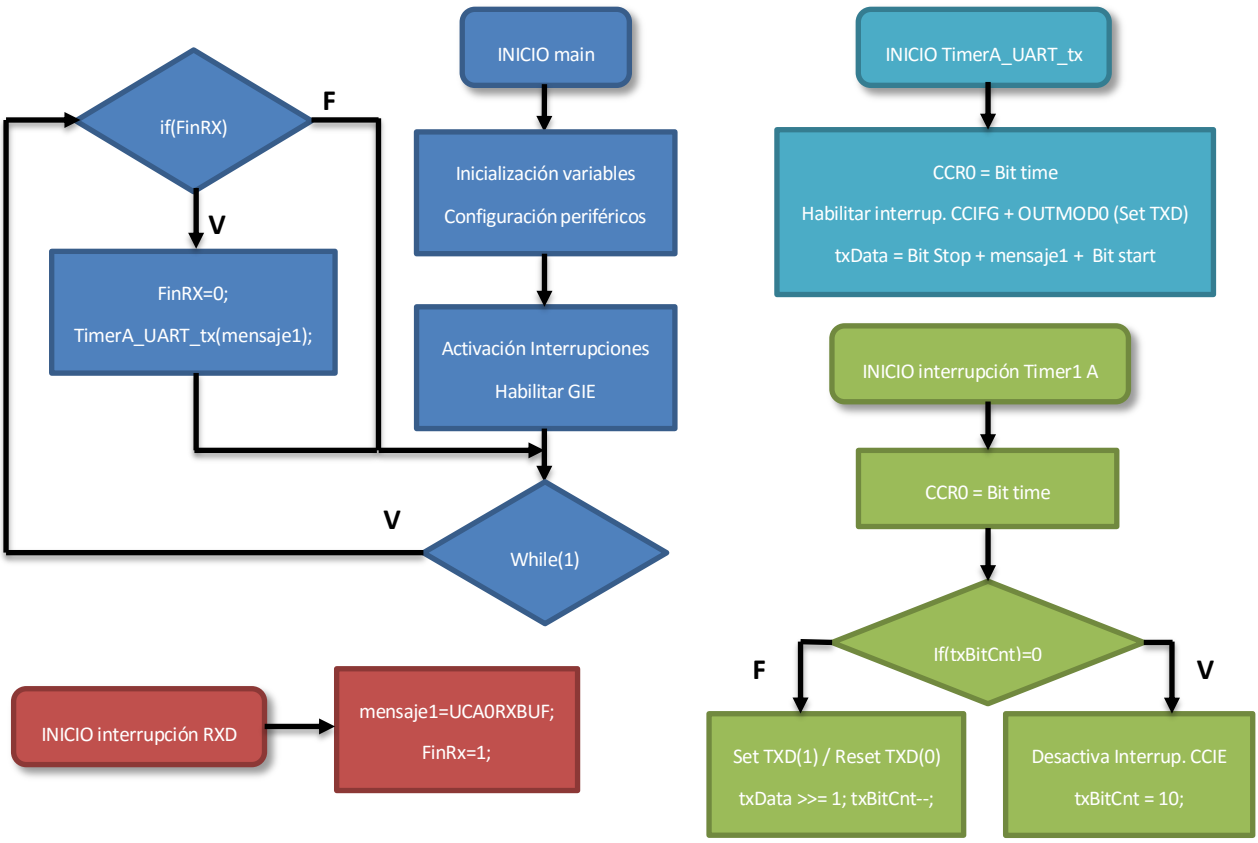

Figura 6-2. Diagrama de flujo programa microcontrolador receptor

6.3 Programación microcontrolador emisor

En el bloque emisor, contamos de nuevo con un microcontrolador MSP430G2553 que tendrá que gestionar, en este caso, el módulo HC-05 emisor y la pantalla VM800. Esto se traduce, básicamente, en la necesidad de programar la USCI_A para trabajar como UART, y la USCI_B para trabajar como SPI de cara a comunicarse con la pantalla. La configuración de la USCI_A para trabajar en modo UART, se hace de manera idéntica al apartado anterior, por lo que no se volverá a incidir en ella. Sí se repasará en detalle, a lo largo de este apartado la configuración de la comunicación SPI y la programación de la interfaz y gestión de la pantalla. Como ya se comentó en el capítulo dedicado a la pantalla VM800, se cuenta como base con las librerías adaptadas por el profesor Manuel Ángel Perales Esteve, que implementan algunos comandos básicos de la pantalla y una serie de funciones Hardware que automatizan el proceso de comunicación entre el microcontrolador y la pantalla que se vio en el capítulo 4 de este proyecto. Será necesario, de todas formas, ampliar algunas funcionalidades de estas librerías para trabajar con elementos como los encoders y los sliders, así como adaptarlas para utilizar el SPI de la USCI_B, para poder utilizar la USCI_A como UART. En el anexo II, se adjuntan todas las librerías y ficheros del programa para poder consultar el código en profundidad.

El orden de las operaciones será similar a lo expuesto en el apartado anterior. Primeramente, se han de llevar a cabo las configuraciones de todos los periféricos que se utilizarán, y posteriormente, se entrará en un bucle while(1) donde se leerán las pulsaciones en la pantalla, o la ausencia de ellas, de cara a dibujar en pantalla los elementos correspondientes. Dentro de este mismo bucle, se aprovecha, en caso de pulsación, para enviar el comando MIDI que corresponda ante la variación de alguno de los controladores que se disponen en pantalla.

6.3.1 Librerías y configuración

En este subapartado, se abordará la configuración de los periféricos y se repasarán las liberías del proyecto. Básicamente, tenemos la librería HC05.h, cuyo principal objetivo es configurar la UART para el módulo HC-05, de la manera en que se vio para el micro receptor, y la librería FT800.h, que define las funciones de alto nivel y comandos necesarios para el manejo de la pantalla. Entre ellas, se encuentran las funciones necesarias para configurar los periféricos del microcontrolador necesarios para la pantalla, como el SPI, y para inicializar la pantalla. Algunas de las funciones más importantes son:

- void **HAL_Configure_MCU**(void): Su principal misión es llevar a cabo la configuración de los periféricos del microcontrolador que se utilizarán a lo largo del programa. Lleva a cabo las configuraciones siguientes:
    1. Desactivar el timer del Watchdog.
    2. Configuración de pines. Ver tabla 6-1 para conocer los pines usados y su función.
    3. Configuración del reloj principal a 8 MHz.
    4. Configuración de USCI_B como SPI de 3 líneas (UCMODE=00; UCSYNC=1) a 500kbps (UCB0BR0=16). UCCKPH a nivel alto (dato se captura el primer flanco y se sobreescribe en el segundo). Transmisión MSB a LSB (UCMSB=1) y microcontrolador como maestro (UCMST=1). La pantalla VM800 será el esclavo en la comunicación.
    5. Habilitación interrupción de llegada de datos en USCI_B (UCB0RXIFG). Activación GIE.
    6. Configuración Timer0 en modo comparación para contar 1 ms. Se utiliza reloj ACLK (TASSEL_1).
- unsigned char **HAL_SPI_ReadWrite**(unsigned char data): Se encarga de cargar en la variable “Buffer_Rx” lo que llegua al registro de entrada UCB0RXBUF. Escribe en UCB0TXBUF el argumento “data” que se quiere enviar. Comunicación bidireccional.

Pin	Función	Explicación
P1.1	RXD UART	Línea de recepción de datos desde el módulo HC-05
P1.2	TXD UART	Línea de transmisión de datos al módulo HC-05
P1.3	Entrada+R_{pullup}	Botón placa Launchpad. Entrada digital con R_{pullup}
P1.5	UCB0CLK	Línea de reloj para comunicación SPI
P1.6	UCB0SOMI	Línea de entrada de datos al Master y salida al Slave
P1.7	UCB0SIMO	Línea de entrada de datos al Slave y salida al Master
P2.1	Salida (Power Down)	Pin para controlar el Power Down de la pantalla
P2.2	Salida (Chip Select)	Controlar Chip Select de SPI por software

Tabla 6-1. Pines utilizados en microcontrolador emisor

- void **HAL_SPI_CSLow**(void), void **HAL_SPI_CSHigh**(void), void **HAL_SPI_PDlow**(void), void **HAL_SPI_PDhigh**(void): El cometido de estas funciones es simplemente actuar sobre los pines de salida Power Down y Chip Select.
- void **espera**(int T): Recibe como argumento el número de milisegundos a esperar. Utiliza para efectuar la espera el Timer0 configurado en "HAL_Configure_MCU". La rutina de interrupción actualiza la variable "t" con el número de milisegundos transcurridos. La función finaliza cuando el valor de ésta llega al argumento "T".
- void **FT800_SPI_SendAddressWR**(dword Memory_Address): Esta función se encarga de enviar la dirección del registro en una operación de escritura. Para ello fuerza los dos primeros bits a ser "10" como se explicó en el capítulo 4. El tipo dword es un define de tipo long. Hay que tener en cuenta que mientras que la dirección es de 3 bytes, el registro de transmisión en la USCI es sólo de 1 byte, por lo que la dirección se va particionando en bytes y mandándose progresivamente con la función "HAL_SPI_ReadWrite" antes mencionada.
- void **FT800_SPI_SendAddressRD**(dword Memory_Address): Hace lo propio de la función anterior, pero para una operación de lectura de registro. Para ello, los dos primeros bits son "00". Como es una lectura, se envía el "dummy" byte necesario al final.
- void **FT800_SPI_Write32**(dword SPIValue32): Escribir un dato de 32 bits en un registro. Siempre necesario enviar antes la dirección del registro con "FT800_SPI_SendAddressWR". Como anteriormente, es necesario mandar cada byte de una sola vez, empezando por el menos significativo.
- void **FT800_SPI_Write16**(unsigned int SPIValue16), void **FT800_SPI_Write8**(byte SPIValue8): Misma función que la anterior, solo que para datos de 16 y 8 bits respectivamente.
- dword **FT800_SPI_Read32**(): Lee un dato de 32 bits desde el FT800, la llamada a esta función devuelve el dato leído. Antes se ha tenido que enviar la dirección del registro con "FT800_SPI_SendAddressRD". De nuevo, se lee byte a byte, empezando por el menos significativo. Es necesario, mientras se lee un dato, enviar bytes de "dummies" por la línea de transmisión, ya que el SPI es bidireccional y siempre escribe mientras lee. Esto no es problema porque la función "HAL_SPI_ReadWrite" puede llamarse con el byte a enviar como argumento y recibir el byte a leer.
- byte **FT800_SPI_Read8**(): Similar a la anterior para un dato de 8 bits. El tipo byte es un char.

- void **FT800_SPI_HostCommand**(byte Host_Command): Esta función se encarga de enviar comandos de Host al FT800. Para llevar a cabo esta tarea, es necesario poner el "Chip Select" a nivel bajo antes de proceder al envío. Posteriormente, se envía un primer byte donde los dos bits MSB han de ser "01" para indicar que se envía un comando. Los 6 bits restantes son el código del comando en cuestión. A continuación, se envían dos "dummy" bytes. Por último, se devuelve el "Chip Select" a nivel alto.
- void **FT800_SPI_HostCommandDummyRead**(void): Esta función esta diseñada para enviar 3 "dummy" bytes de manera similar a las anteriores funciones. Esto es necesario en la configuración de la pantalla para encenderla.
- unsigned int **FT800_IncCMDOffset**(unsigned int Current_Offset, byte Command_Size): La misión de esta función es calcular el offset necesario con el que hay que escribir el próximo comando en la cola de comandos del FT800. Se le pasan como argumentos el offset actual y el tamaño del comando que se quiere enviar. Devuelve el nuevo offset donde se escribirá el comando, que es resultado de la suma de ambos argumentos. Es necesario volver al principio de la cola de comandos cuando se llega al valor 4095 (la cola tiene un tamaño de 4kB).
- void **EscribeRam32**(unsigned long dato): Automatiza el proceso de escritura en un registro de la FIFO, manejando el Chip Select para comenzar y terminar la comunicación, y llamando a la función "FT800_SPI_SendAddressWR" para mandar primero la dirección donde se va a escribir, y seguido el dato con la función "FT800_SPI_Write32". Por último, se utiliza la función "FT800_IncCMDOffset" para calcular el offset necesario para el próximo comando.
- void **EscribeRam16**(unsigned int dato), void **EscribeRam8**(char dato): Misma funcionalidad que las anteriores, pero para datos de 16 y 8 bits respectivamente.
- void **EscribeRamTxt**(char *cadena): Permite enviar una cadena de caracteres acabada en 0.
- void **Comando**(unsigned long COMM): El cometido de esta función es enviar un comando de 4 bytes. Es similar a "EscribeRam32", pero menos flexible. Es útil para enviar comandos como "CMD_DISPLAY" que siempre van a tener el mismo tamaño fijo de 4 bytes y no requieren parámetros adicionales.
- void **PadFIFO**(void): Como se ha comentado anteriormente, es necesario cuando enviemos un parámetro que el número de bytes que éste ocupe sea 4 o múltiplo de 4. Como esto no se cumple en ocasiones, es necesario enviar "dummy" bytes hasta conseguir ese objetivo. Esto se consigue llamando a la función PadFIFO, que envía bytes de 0 mediante la función "EscribeRam8" hasta que se cumpla tal condición. En general, es posible hacerlo manualmente calculando los bytes que hacen falta y mandándolos sin necesidad de esta función. Sin embargo, algunos comandos aceptan como parámetro cadenas de caracteres que no tienen un tamaño fijo, por lo que es imposible predecir cuantos bytes adicionales hará falta enviar. Siendo entonces necesario ejecutar esta función al final.
- void **Dibuja**(void): Da la orden de ejecutar la lista de comandos enviada hasta el momento. Para ello, envía el comando CMD_DISPLAY que cierra la lista e ignora los nuevos comandos que lleguen. Posteriormente, se envía el comando CMD_SWAP que cambia la lista actual y crea una nueva donde se podrán recibir nuevos comandos. Por último, se ejecuta la función "Ejecuta_Lista".
- void **Ejecuta_Lista**(void): Escribe en la posición de memoria "REG_CMD_WRITE" el valor del offset actual para recuperarlo cuando sea necesario volver a escribir más comandos. Tras esto, el FT800 comienza la ejecución de la cola circular de comandos desde la posición "REG_CMD_READ" hasta topar con la posición "REG_CMD_WRITE". Los nuevos comandos de la nueva lista creada se pueden ir escribiendo a partir de la última dirección.
- void **ComEsperaFin**(void): Espera la finalización de la ejecución de todos los comandos de la lista. Para ello, se leen los registros de la cola "REG_CMD_WRITE" y "REG_CMD_READ" y se espera hasta que estos sean coincidentes.

- void **Inicia_pantalla**(void): Esta función es ejecutada al principio del programa, tras la configuración de los propios periféricos del microcontrolador mediante la función "HAL_Configure_MCU**",** y lleva a cabo todas las configuraciones necesarias para encender e inicializar la pantalla. En esta función, se hace uso de los comandos de Host, que se pueden enviar con "FT800_SPI_HostCommand". Básicamente, se siguen los siguientes pasos [14]:
    1. Poner pin de Power Down a nivel alto para encender la pantalla.
    2. Configuración del oscilador del FT800. Para ello se mandan comandos de Host CLKEXT(0x44) y CLK48M(0x62), que habilitan el oscilador y lo configuran a 48 MHz.
    3. Se resetean todos los registros del FT800 mediante el comando de Host CORERST(0x68).
    4. Asegurar que el FT800 se encuentra en estado activo mediante el envío del comando de Host ACTIVE(0x00). Es equivalente a hacer una lectura de la posición de memoria 0.
    5. Como medida extra de seguridad del correcto funcionamiento, se lee el Chip ID del FT800 que debe de ser 0x7C en todo caso.
    6. A continuación, es necesario ajustar el funcionamiento del FT800 al display que manejará. Para ello, se escriben los valores adecuados para los registros del display. Por ejemplo, se ha de configurar el tamaño de la pantalla (320x240), escribiendo en la dirección correspondiente de los registros REG_VSIZE y REG_HSIZE, los valores de la resolución de pantalla.
    7. Configuración de los registros asociados al control de la pantalla táctil. En particular, se configura el threshold de la pantalla.
    8. Pintar primera pantalla de color negro. Enviar comandos "CMD_DISPLAY" y "CMD_SWAP" para cerrar la lista de comandos actual y crear una nueva.
    9. Acceder a los puertos de E/S del FT800 para habilitar el display.
- void **Nueva_pantalla**(unsigned long int color): Esta función crea una nueva pantalla con el color de fondo el color que se pasa como argumento (número de 24 bits con formato RGB). Para ello, espera la ejecución de la lista anterior mediante "ComEsperaFin". A continuación, enviar el comando "CMD_DLSTART" para comenzar una lista nueva.
- void **Lee_pantalla**(void): Se encarga de leer la posición de la pulsación en la pantalla. La posición se guarda en el registro de 32 bits "REG_TOUCH_SCREEN_XY", con los 16 bits MSB para la posición en el eje X y los 16 bits LSB para la posición en el eje Y. Se lee el registro y se almacenan los bits correspondientes al eje X en POSX y al eje Y en POSY. Es conveniente ejecutar esta función al comienzo del bucle principal, para llevar cabo el dibujo de los elementos en función de la pulsación.
- void **Espera_pant**(void): Mantiene un "Lee_pantalla" hasta que se lleva a cabo una pulsación. Es útil para mantener pantallas hasta que se seleccione una opción que pasa a una nueva pantalla.
- void **Calibra_touch**(void): Lleva a cabo una calibración de la pantalla. Se utiliza el comando "CMD_CALIBRATE" que ejecuta la rutina de calibración de la pantalla, que se basa en pedir al usuario tocar la pantalla en 3 puntos concretos.

Hasta ahora, se han resumido las funciones más importantes para la configuración y el funcionamiento más básico de la pantalla. Con lo expuesto hasta ahora, se podría llevar cabo la estructura básica de un programa que permita manejar la pantalla VM800. Sin embargo, es necesario ahondar en los comandos que tenemos a nuestra disposición para pintar objetos en la pantalla, y en las posibles estrategias que se han de seguir para efectuar toda una lista de comandos para pintar una pantalla con diversos elementos con los que se pueda interactuar y modificar la pantalla ante tales acciones.

6.3.2 Comandos del coprocesador y estructura básica del programa

Efectuadas todas las configuraciones, y una vez estamos dentro del bucle principal del programa, es conveniente siempre preguntar por la posición de la pulsación, por si se hubiera producido. En cualquier caso, se almacena el registro con los datos de la posición en las variables POSX y POSY. Esto nos permitirá discernir si se produjo una pulsación, y en caso afirmativo, actuar en consonacia a tal situación.

Una vez tenemos esa información, es conveniente empezar con la función "Nueva_pantalla", que creará un fondo del color igual al argumento que pasemos a la función. A partir de aquí, basta con enviar todos los comandos necesarios para ir estableciendo distintos elementos a lo largo de la pantalla.

Para los más sencillos y que tengan tamaño fijo, basta con utilizar la función "Comando" anteriormente explicada para mandar el comando. En otros casos, los comandos, a parte del código de 32 bits que lo representa, es necesario añadir datos adicionales que están relacionados con diversos parámetros del elemento a dibujar. En este caso, es conveniente realizar funciones, que se verán posteriormente, dedicadas a mandar cada tipo específico de estos comandos, que prevean el envío a parte del código del comando, los datos adicionales requeridos, y en caso de que fuera necesario, el envío de "dummy" bytes para ocupar un número de bytes múltiplo de 4.

En buena parte de estos objetos, estos no son solamente un simple elemento dibujado en pantalla, sino que permiten la interacción con los mismos. Por ejemplo, un botón puede ser pulsado, cambiando la representación de efecto 3D a efecto planto. Un slider, puede ser pulsado y desplazado en una dirección. Este cambio también se puede representar en pantalla.

La estrategia a seguir en estos casos, y teniendo en cuenta que al principio del bucle se ha leído la posición de la pulsación, y se ha almacenado en POSX y POSY, es preguntar si se ha pulsado en el área que ocupa el elemento en cuestión. Si esto no se ha producido, se pinta el objeto con efecto 3D en su posición habitual sin efectuar ningún cambión. En caso afirmativo, se pintará ese mismo objeto con efecto de pulsación, en su nueva posición si fuera necesario, y efectuando las acciones que se vean oportunas. Por ejemplo, en este proyecto, se aprovechará este momento para enviar el mensaje MIDI adecuado al interactuar con alguno de los controladores que se disponen en pantalla. Por último, al final del bucle, se llama a la función Dibuja(), para acabar la lista actual con todos los comandos de la pantalla que se quiere plasmar. Esta acción dará la orden al FT800 de ejecutar esta lista con todos los cambios que se han producido en los objetos de pantalla en esta pasada del bucle. Todo este proceso se sigue repitiendo en cada ocasión que se ejecuta el bucle, de manera que se vayan recogiendo todos los cambios producto de la acción del usuario sobre la pantalla. Un ejemplo de lo anterior sería

```
while(1){
  Lee_pantalla(); //Leer variables POSX y POSY
  ...
  if(POSX>170 && POSX<297 && POSY>15 && POSY<35) //Pulsacion en el slider
     {Vp1=POSX-170; //Almacenar posicion relativa, restando la coordenada X inicial
      ComSlider(170,15,127,20,256,Vp1,127); //Dibujo slider pulsado (sin efecto 3D)
      ...} //Ejecucion de instrucciones relativas a pulsación
  else{ComSlider(170,15,127,20,0,Vp1,127);} //Dibujo slider sin pulsar (efecto 3D)
  ...
  Dibuja(); //Ejecutar lista
}
```

En el ejemplo anterior se hace uso de la función "ComSlider" que dibuja un slider a partir de los parámetros que se pasan como argumentos. En este caso, es un slider horizontal de 127 valores y ancho de 20 píxeles. La variable Vp1 almacena la posición relativa del slider y se actualiza con la posición de la pulsación en caso de que ésta se encuentre dentro de la superficie del slider. En caso contrario, Vp1 permanece constante y no se produce ningún cambio en el slider.

Descrita la manera de utilizar estos comandos, se procederá a describir las funciones que se encargan de implementar algunos de los más relevantes. Para construir estas funciones, es conveniente consultar la "FT800 Programmer Guide" [15], donde se describe la funcionalidad de estos comandos, los 4 bytes que los identifican y diferencian del resto, y todos los datos que es necesario enviar a posteriori, con el número de bits que ocupan, para una correcta representación del objeto. Para facilitar la labor, se disponen en las librerías de defines que asocian el código de 4 bytes del comando a una cadena de caracteres con el nombre del comando. Por ejemplo, CMD_ROTATE es el comando 0xFFFFFF29 (4294967081), y en la librería se asocia:

```
#define CMD_ROTATE              4294967081
```

Conocido esto, se pueden abordar las funciones de comandos:

- void **ComVertex2ff**(int x,int y): Establece un vértice de coordenadas x,y que servirá para pintar un elemento. Éste dependerá de comandos que se hayan ejecutado anteriormente. Si anteriormente se mandó el comando "CMD_BEGIN_RECTS" (se puede utilizar la función comando() para enviarlo), este vértice corresponderá a un rectángulo que necesitará de otro vértice para ser completado. De esta manera, cada pareja de vértices consecutivos define un rectángulo que se representa en pantalla. Si se envió el comando "CMD_BEGIN_LINES", cada pareja de vértices definirá una línea. Tanto RECTS como LINES, son en realidad operaciones "primitivas" del comando BEGIN. Éstas se denominan así porque modifican la acción del comando "ComVertex2ff", sin tener una representación directa propia. La influencia de estas primitivas, acaba con el comando "CMD_END". A partir de ahí, es necesario asignar otra primitiva para dibujar elementos con vértices.

  Tanto rectángulos como líneas pueden ver afectado el grosor del trazo mediante el comando "CMD_LINEWIDTH". Se puede utilizar llamando a la función comando de la manera:

  ```
  Comando(CMD_LINEWIDTH+L);
  ```

  Siendo L el ancho en número de píxeles entre 16.

- void **ComColor**(int R, int G, int B): Aplica el comando COLOR_RGB, de cara a modificar el color de los elementos que se dibujen posteriormente. Se mandan 4 bytes, siendo el primero el código del comando (0x04) y los 3 restantes los bytes que recibe como argumento con el color. Existen otras dos opciones para cambiar el color de un elemento. Dependiendo del objeto que se pinte, se podrá usar uno u otro, o varios a la vez para cambiar el color de distintos componentes del objeto. Para ello, tenemos los comandos: CMD_FGCOLOR y CMD_BGCOLOR. El funcionamiento es el mismo que COLOR_RGB. Para CMD_FGCOLOR tenemos la función ComFgcolor.

- void **ComTXT**(int x, int y, int fuente, int ops, char *cadena): Esta función permite representar cuadros de texto en pantalla. Para ello, se envía el comando CMD_TEXT(0xFFFFFF0C). Adicionalmente, es necesario enviar como parámetros:

  1. Int x. Coordenada x esquina superior izquierda. (2 bytes).
  2. Int y. Coordenada y esquina superior izquierda. (2 bytes).
  3. Int fuente. Fuente del texto. Valor entre 0 y 31. (2 bytes).
  4. Int ops. Elección de opciones: OPT_CENTERX (centra el texto horizontalmente), OPT_CENTERY (centra el texto verticalmente), OPT_CENTER (centra el texto en ambas direcciones), OPT_RIGHTX (texto justificado a la derecha). (2 bytes)
  5. Char cadena. Cadena de caracteres con el tecto a enviar. Importante acabarla en 0 siempre. (número de bytes dependiendo de la longitud del texto).

  Para enviar los datos, se utilizan las funciones EscribeRamx, dependiendo del tamaño de bytes. Para enviar la cadena de caracteres, se envía de uno en uno con EscribeRam8, hasta el final de la cadena. Necesario enviar un 0 final con EscribeRam8. Como el número de bytes total enviado en el comando es variable, necesario acabar la función con PadFIFO().

- void **ComButton**(int x, int y, int w, int h, int font, int ops, char *cadena): Permite dibujar un botón mediante el comando CMD_BUTTON (0xFFFFFF0D). Tiene, además, como parámetros:
    1. Int x. Coordenada x esquina superior izquierda. (2 bytes).
    2. Int y. Coordenada y esquina superior izquierda. (2 bytes).
    3. Int w. Ancho del botón en píxeles. (2 bytes).
    4. Int h. Alto del botón en píxeles. (2 bytes).
    5. Int font. Fuente del texto dentro del botón. (2 bytes).
    6. Int ops. Elección de opciones: OPT_FLAT(256) para efecto plano, 0 para efecto 3D. (2 bytes).
    7. Char cadena. Cadena de caracteres con texto para incluir dentro del botón.

    Para implementar esta función, se hace de forma idéntica a la función anterior, con EscribeRam para enviar los datos y finalizando con PadFIFO.
- void **ComSlider**(int x, int y, int w, int h, int options, int val, int range): Se encarga de dibujar una barra deslizadora mediante el comando CMD_SLIDER (0xFFFFFF10). Los parámetros son:
    1. Int x. Coordenada x de la esquina superior izquierda del slider. (2 bytes).
    2. Int y. Coordenada y de la esquina superior izquierda del slider. (2 bytes).
    3. Int w. Ancho o dimensión horizontal del slider. (2 bytes).
    4. Int h. Altura o dimensión vertical del slider. (2 bytes). Si h es mayor que w la barra deslizadora es vertical. En caso contrario, será horizontal.
    5. Int options. Elección de opciones: OPT_FLAT(256) para efecto plano, 0 para efecto 3D. (2 bytes).
    6. Int val. Es el valor de posición relativa del slider. (2 bytes). Tiene que estar dentro del rango.
    7. Int range. Es el rango de valores que acepta el slider. (2 bytes). No tiene porque coincidir con las dimensiones del slider.

    Como se puede observar, el comando posee un número de bytes total fijo que no es múltiplo de 4. Es necesario acabar con EscribeRam16(0) para que esto suceda.
- void **ComDial**(int x, int y, int r, int options, int val): Permite dibujar controles rotatorios con el comando CMD_DIAL (0xFFFFFF0D). Los parámetros son los siguientes:
    1. Int x. Coordenada x del centro del círculo. (2 bytes).
    2. Int y. Coordenada y del centro del círculo. (2 bytes).
    3. Int r. Radio del círculo en píxeles. (2 bytes).
    4. Int options. Elección de opciones: OPT_FLAT(256) para efecto plano, 0 para efecto 3D. (2 bytes).
    5. Int val. Posición del control rotatorio. Valor entre 0 y 65 355. (2 bytes). Para el valor 0, el control se encuentra en posición vertical hacia abajo, en 0x4000 se encuentra en el eje horizontal hacia la izquierda, en 0x8000 en posición vertical hacia arriba, y en 0xC000 se encuentra hacia la derecha. Estos valores se pueden usar como referencia a la hora de obtener valores intermedios. Es necesario crear otra función que permita conocer el valor que hay que pasar a la función en “int val” para colocar el control correctamente a partir de los valores de POSX y POSY. Ésta es la función “theta”.

    Necesario, de nuevo, acabar la función con EscribeRam16(0) para que número de bytes sea múltiplo de 4.

- unsigned int **Theta**(int x,int y): La misión de esta función es devolver el valor de 16 bits de la posición de un control rotatorio, tomando como argumentos la posición relativa a ejes trasladados al centro del control. De esta manera: x=POSX-CoordX_{centro}; y=POSY-CoordY_{centro}.

  Con estos valores, se comprueba según el signo de ambos la posición de la pulsación en alguno de los 4 cuadrantes del círculo. Una vez conocido el cuadrante, se puede calcular la tangente dividiendo y/x. El círculo se divide en sectores en función del ángulo, asociándoles el valor adecuado para el encoder rotatorio. Contra más sectores, más fino será el control. Con el valor de la tangente se asocia la pulsación al sector correcto y se devuelve el valor correspondiente.

- unsigned int **Encoder**(int Th): Conocido el valor del encoder (Th), se toma como argumento y se devuelve el valor MIDI correspondiente entre 0 y 127 para ser enviado.

Existen un mayor número de comandos del coprocesador que permiten dibujar otros objetos, así como comandos asociados al sintetizador de audio, entre otros. En este caso, se han descrito los comandos que se van a utilizar para el proyecto. Los demás se pueden construir de manera análoga a lo visto.

6.3.3 Diseño de la interfaz

En los anteriores subapartados se han repasado todos los conceptos y todas las funciones y comandos necesarios para poder diseñar una pantalla. Así mismo, se ha descrito la estructura básica que ha de seguir el código para manejar la pantalla. Conocido todo esto, se abordará el diseño de la interfaz que se implementará en este proyecto para el controlador MIDI.

El objetivo es establecer una primera pantalla de presentación, que espere a la pulsación del usuario antes de presentar la interfaz del controlador. Una vez pasada esta primera pantalla, se entraría en el bucle principal, en el que se implementaría la interfaz. Esta interfaz estará formada por tantas pantallas como canales MIDI se implementen, en cuada una de las cuales se dispondrán los controladores específicos de cada canal, cuyos valores se han de mantener en memoria cuando se pase a otra pantalla. Esto implica multiplicar tantas veces como números de pantallas hay, la estructura de comandos que implementa la pantalla. Para ello, se dispone una estructura de sentencias "ifs" que preguntan por el canal actual, y pintan la pantalla con los controladores en los valores adecuados del canal. Debido a la limitación de memoria en el dispositivo, se limitará el control a los 4 primeros canales de MIDI, de los 16 posibles.

Para establecer el canal, se define la variable "Channel". De cara a la selección del mismo, se dispone un selector con dos botones en la esquina superior, y un recuadro en medio de los dos botones que informa del canal actual. De esta manera, el botón de la izquierda decrementa el canal en una unidad (valor mínimo del canal es 1), y el botón de la derecha incrementa el canal en otra unidad (máximo canal es 4). El selector estará presente en todas las pantallas independientemente del canal.

De este modo, para cada canal, se disponen sus propios controles. En este caso, los parámetros MIDI a controlar serán los siguientes:

- **Volume** (CC#7): Barra deslizadora horizontal de rango 0-127, que son los valores típicos de un dato MIDI. Longitud de 127 píxeles y ancho de 20 píxeles.
- **Balance** (CC#8): Barra deslizadora horizontal de rango 0-127. Dimensiones 127x20.
- **Pitch Bend** (Mensaje de canal): Slider vertical de rango 0-127. Dimensiones 127x20.
- **Modulation** (CC#1): Barra deslizadora vertical de rango 0-127. Dimensiones 127x20.
- **Breath** (CC#2): Barra deslizadora vertical de rango 0-127. Dimensiones 127x20.
- **Pedal Sustain** (CC#64): Encoder rotatorio de radio 35 píxeles.
- **Pan** (CC#10): Encoder rotatorio de radio 35 píxeles.
- **Program Change** (Mensaje de canal): Implementado con dos botones que varían el "program".

Estos controles se dispondrán a lo largo de la pantalla de la siguiente manera, de forma aproximada:

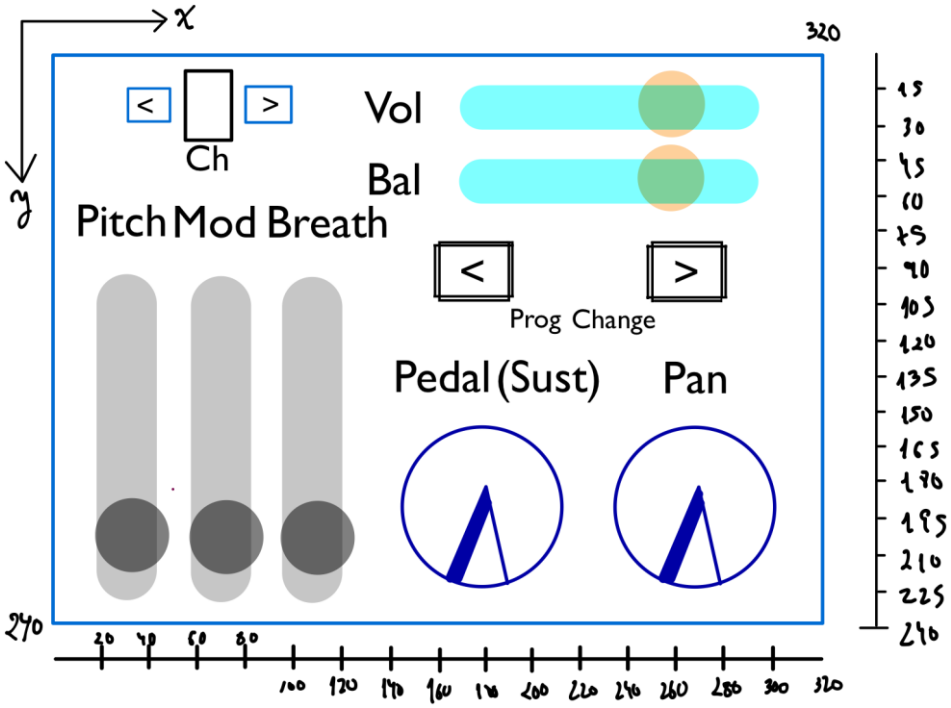

Figura 6-3. Interfaz del controlador MIDI

El código del programa sigue la estructura desarrollada en apartados anteriores. Todos los controles se dibujan por defecto con efecto 3D. Si se pulsa alguno en particular, el efecto cambia a pulsado. En función de la pulsación, se actualiza la posición del controlador. En el caso de los sliders, estos van a su nueva posición, dentro del rango de 0 a 127. Simplemente es necesario hacer una traslación de la posición en el eje de la dirección del slider a la posición 0 del mismo, para obtener el nuevo valor de posición del slider con el que se ha de pintar nuevamente. En el caso de los encoders rotatorios, el funcionamiento es algo más complicado. Es necesario llamar a la función "Theta" explicada anteriormente con la posición relativa de la pulsación a los ejes del círculo como argumentos. La función "Theta" nos proporciona el valor necesario para pintar el encoder en su nueva posición. Sin embargo, necesitamos un valor entre 0 y 127 para actualizar el parámetro MIDI. Para ello, llamamos a la función "Encoder" pasando el valor de la posición como argumento, obteniendo como resultado el valor dentro del rango de entre 0 y 127 necesario para los comandos MIDI. Por último, en el caso de los botones, estos simplemente actualizan su efecto y actúan sobre una variable incrementándola o decrementándola.

6.3.4 Envío de mensajes MIDI

Una vez se actualiza el controlador a su nuevo estado por pantalla, es necesario enviar a través de la UART el mensaje MIDI con el parámetro actualizado a su nuevo valor. Para ello, dentro del mismo "if" donde se pregunta por la pulsación o no pulsación dentro del área del controlador en cuestión, se lleva a cabo el correspondiente envío, justo después de enviar el comando correspondiente al FT800 para actualizar el controlador.

Como se explicó en el segundo capítulo, un mensaje MIDI tiene básicamente un byte de estado, y hasta dos bytes de datos. De esta manera, utilizamos una variable unsigned char llamada "comando" donde se escribe el byte a enviar en cada caso, empezando por el byte de estado. Posteriormente, se carga el byte comando en el registro UCA0TXBUF donde el byte en cuestión será transmitido por la línea TXD al módulo Bluetooth HC-05. Éste, si esta enlazado con el módulo receptor, se encargará de transmitirlo al bloque receptor.

Tras cargar el byte en el registro UCA0TXBUF se hace una espera mediante "__delay_cycles(n)", siendo el argumento "n" el número de ciclos de reloj a esperar. Tras la espera, se carga el data byte 1 y se repite el proceso. En caso de que sea necesario, se carga el data byte 2 y se hace lo propio. Estas esperas intermedias entre el envío de los bytes, permite a parte de asegurar el envío del comando anterior antes de volver a escribir en el búfer TXD, que la pulsación de un botón no incremente la variable asociada de manera muy rápida. Facilitando el control de los comandos.

Como se vio en el segundo capítulo, estos bytes de estado y de datos, cambian su estructura en función del mensaje. A continuación, se verán los correspondientes a los controles que se utilizan en este proyecto:

- **Volume** (CC#7): Control de barra deslizadora.
    1. Status Byte: Control Change. Estructura: 0xBn (1011nnnn). Siendo n el canal MIDI.

       Para el canal 1: 0xB0. Canal 2: 0xB1. Canal 3: 0xB2. Canal 4: 0xB3.
    2. Data Byte 1: Nº de Control Change. En este caso es CC#7. Para todos los canales: 0x07.
    3. Data Byte 2: Valor del parámetro entre 0 y 127. Se envía directamente la variable de posición del controlador que se utiliza para pintar el slider.
- **Balance** (CC#8): Control de barra deslizadora.
    1. Status Byte: Control Change. Estructura: 0xBn (1011nnnn). Siendo n el canal MIDI.

       Para el canal 1: 0xB0. Canal 2: 0xB1. Canal 3: 0xB2. Canal 4: 0xB3.
    2. Data Byte 1: Nº de Control Change. En este caso es CC#8. Para todos los canales: 0x08.
    3. Data Byte 2: Valor del parámetro entre 0 y 127. Se envía directamente la variable de posición del controlador que se utiliza para pintar el slider.
- **Pitch Bend**: Control de barra deslizadora, con retorno a posición inicial tras soltar.
    1. Status Byte: Mensaje de canal Pitch Bend. Estructura: 0xEn (1110nnnn). Siendo n el canal MIDI. Para el canal 1: 0xE0. Canal 2: 0xE1. Canal 3: 0xE2. Canal 4: 0xE3.
    2. Data Byte 1: Byte LSB del valor del Pitch. En este caso, vector de 0, porque sólo se controlará con los 127 valores del byte MSB. Para todos los canales: 0x00.
    3. Data Byte 2: Byte MSB del valor del Pitch. Para todos los canales, se envía el valor de la variable de posición del slider (0-127).
- **Modulation** (CC#1): Control de barra deslizadora.
    1. Status Byte: Control Change. Estructura: 0xBn (1011nnnn). Siendo n el canal MIDI.

       Para el canal 1: 0xB0. Canal 2: 0xB1. Canal 3: 0xB2. Canal 4: 0xB3.
    2. Data Byte 1: Nº de Control Change. En este caso es CC#1. Para todos los canales: 0x01.
    3. Data Byte 2: Valor del parámetro entre 0 y 127. Se envía directamente la variable de posición del controlador que se utiliza para pintar el slider.
- **Breath** (CC#2): Control de barra deslizadora.
    1. Status Byte: Control Change. Estructura: 0xBn (1011nnnn). Siendo n el canal MIDI.

       Para el canal 1: 0xB0. Canal 2: 0xB1. Canal 3: 0xB2. Canal 4: 0xB3.
    2. Data Byte 1: Nº de Control Change. En este caso es CC#2. Para todos los canales: 0x02.
    3. Data Byte 2: Valor del parámetro entre 0 y 127. Se envía directamente la variable de posición del controlador que se utiliza para pintar el slider.

- **Pedal Sustain** (CC#64): Encoder rotatorio.
    1. Status Byte: Control Change. Estructura: 0xBn (1011nnnn). Siendo n el canal MIDI.

       Para el canal 1: 0xB0. Canal 2: 0xB1. Canal 3: 0xB2. Canal 4: 0xB3.
    2. Data Byte 1: Nº de Control Change. En este caso es CC#64. Para todos los canales: 0x40.
    3. Data Byte 2: Valor del parámetro entre 0 y 127. Se envía el valor que devuelve la función "Encoder" a partir del valor de la posición del controlador.
- **Pan** (CC#10): Encoder rotatorio.
    1. Status Byte: Control Change. Estructura: 0xBn (1011nnnn). Siendo n el canal MIDI.

       Para el canal 1: 0xB0. Canal 2: 0xB1. Canal 3: 0xB2. Canal 4: 0xB3.
    2. Data Byte 1: Nº de Control Change. En este caso es CC#10. Para todos los canales: 0xA.
    3. Data Byte 2: Valor del parámetro entre 0 y 127. Se envía el valor que devuelve la función "Encoder" a partir del valor de la posición del controlador.
- **Program Change**: Botón a la izquierda disminuye la variable "Program" en 1 hasta un mínimo de 0. El botón a la derecha aumenta la variable en 1 hasta un máximo de 127.
    1. Status Byte: Mensaje de canal Program Change. Estructura: 0xCn (1100nnnn). Siendo n el canal MIDI. Para el canal 1: 0xC0. Canal 2: 0xC1. Canal 3: 0xC2. Canal 4: 0xC3.
    2. Data Byte 1: Valor del parámetro entre 0 y 127. Se envía la variable "Program".

    No es necesario un segundo byte de datos para un Program Change.

Enviado el mensaje MIDI al completo, se termina la lista de comandos para la pantalla actual y se llama a la función Dibuja() para finalizar la pantalla y pasar a una nueva en la próxima pasada del bucle, donde se vuelve a comprobar la pulsación y pintar de nuevo los elementos actualizados.

En este apartado, se ha realizado una exposición esquemática del programa del microcontrolador emisor. Para mayor detalle, consultar el anexo II.

7 CONEXIÓN FÍSICA DE LOS COMPONENTES

Descrita la implementación del proyecto a nivel de software, se dedicará el presente capítulo a especificar las conexiones físicas entre los distintos componentes que conforman el proyecto a nivel de hardware. Para ello, nos basaremos en el conocimiento de los distintos periféricos y pines a utilizar que se han descrito en los capítulos anteriores. Adicionalmente, se incorporará una breve guía de usuario para la utilización del dispositivo.

7.1 Conexionado de los componentes

Este apartado se dedicará a la conexión del hardware. Por un lado, tenemos el bloque receptor que se conecta a un módulo HC-05 y a una placa de interconexión MIDI, que posee dos conectores, uno MIDI IN, y otro MIDI OUT para poder conectar la salida UART del microcontrolador a partir de los pines que ofrece la placa. Por otro lado, el bloque emisor, con otro microcontrolador MSP430G2553 que se conecta a la pantalla VM800 y a otro módulo HC-05.

La conexión entre ambos bloques es inalámbrica mediante tecnología Bluetooth. El enlace entre ambos módulos HC-05 se realiza inmediatamente si tienen alimentación y se han llevado a cabo las configuraciones necesarias mediante comandos AT, tal y como aparece en el capítulo 3.

7.1.1 Conexiones bloque receptor

En el microcontrolador del bloque receptor, se utilizan los pines P1.1 (RXD) y P1.2 (TXD) para la recepción mediante la UART hardware desde el módulo Bluetooth. Además, es necesario conectar la salida de la UART software (P2.0) al pin "Out" de la placa de interconexión MIDI, que implementa toda la circuitería necesaria para un conector MIDI. A través del puerto MIDI OUT de esta placa, se puede conectar a cualquier dispositivo MIDI por su puerto MIDI IN y recibir los distintos comandos. En este caso, se utilizará un cable MIDI-USB para utilizar herramientas software en un PC para recibir los comandos MIDI. El esquema básico de conexión sería el siguiente:

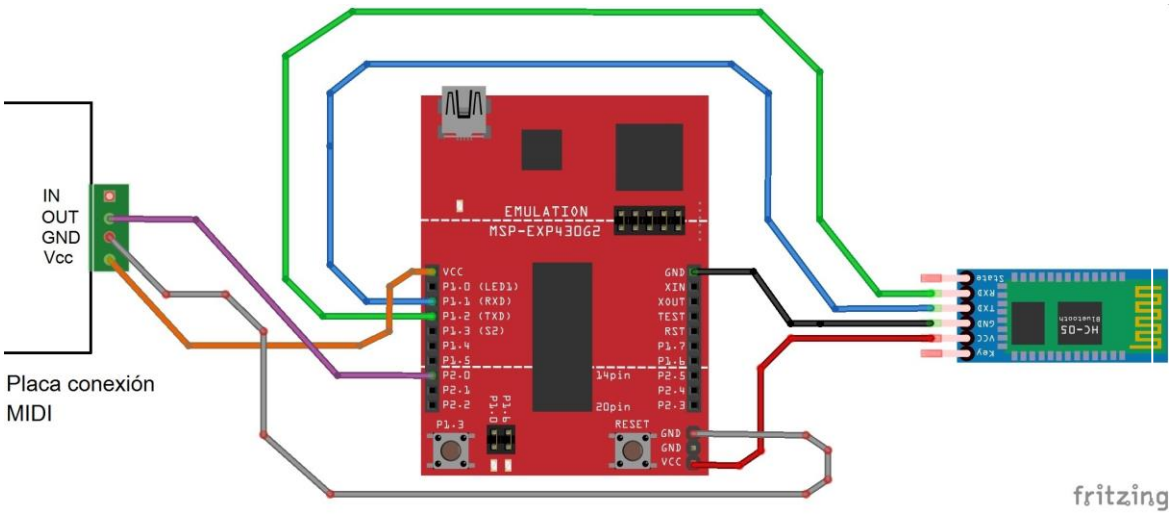

Figura 7-1. Esquema de conexionado del bloque receptor

7.1.2 Conexiones bloque emisor

Para el bloque emisor, se pueden consultar los pines utilizados del microcontrolador en la tabla 6-1. La conexión del módulo HC-05 se efectúa de la misma manera que en el receptor. Para la conexión de la pantalla, consultar los pines de la misma en la tabla 4-2. Se presenta a continuación un esquema básico del conexionado:

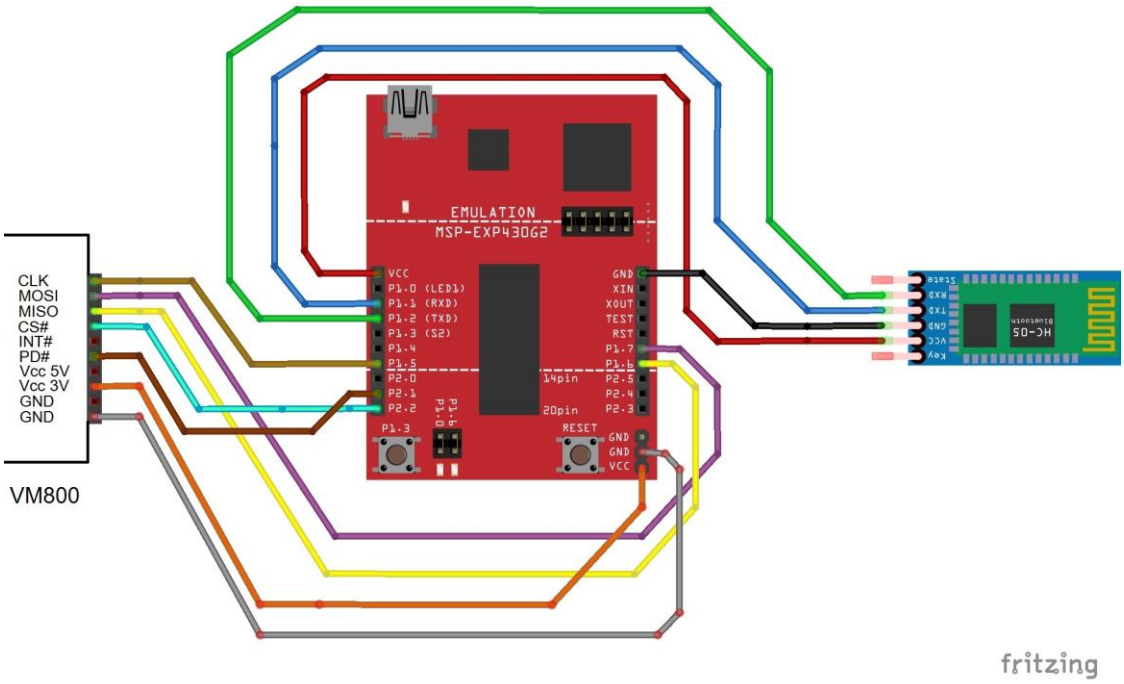

Figura 7-2. Esquema de conexionado del bloque emisor

7.2 Guía de usuario

Para poder comprobar el funcionamiento del controlador MIDI, es necesario, primero de todo, llevar cabo la configuración de los módulos HC-05 como se especifica en el capítulo 3, para asegurar un enlace correcto de los módulos y la compatibilidad con la UART de los microcontroladores.

A continuación, se han de llevar a cabo las conexiones que se muestran en el apartado anterior. El enlace de los módulos Bluetooth será automático una vez tengan alimentación. El Puerto MIDI OUT de la placa de conexión MIDI se conecta a otro conector MIDI IN. En general, puede ser cualquier dispositivo del mercado que tenga entrada MIDI. En este caso, se utilizará un cable MIDI-USB para poder utilizar el controlador con un PC. Al realizar la conexión con el PC, éste detectará el dispositivo y lo configurará.

Desde el PC, será necesario arrancar el programa con el que se desee trabajar. Para un correcto funcionamiento, será siempre necesario elegir el dispositivo “MIDI-USB” que se ha conectado como MIDI IN. Adicionalmente, se puede elegir el sintetizador de Windows “Microsoft GS Wavetable Synth” o alguno propio que incorpore el software, para poder reproducir las secuencias MIDI que se creen. En este caso, se utilizarán los programas “MIDI-OX” y “Virtual MIDI Piano Keyboard”. El primero es especialmente útil para monitorizar todos los comandos que llegan por el puerto MIDI IN, aunque también puede mapearse el sintetizador de Windows como puerto MIDI OUT. El segundo, posee funcionalidades parecidas, pero ofrece una interfaz mucho más intuitiva, con una serie de teclas asociadas a las distintas notas MIDI. Posee además la posibilidad de representar diferentes controles MIDI y poder ver como evolucionan los distintos parámetros que se controlan desde la interfaz del controlador, así como actuar sobre ellos directamente desde el propio programa.

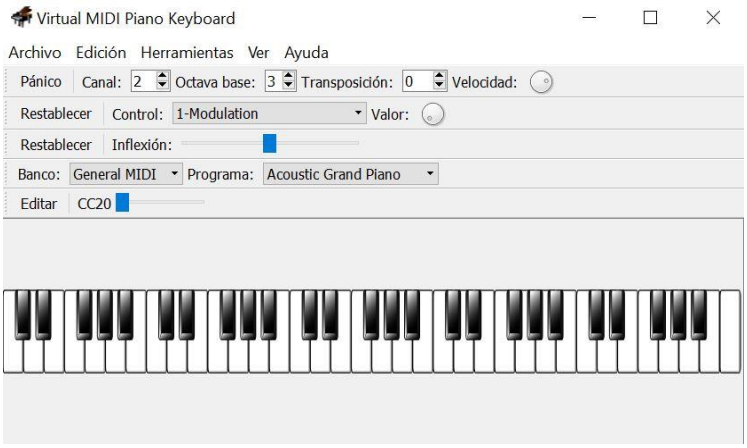

Figura 7-3. Virtual MIDI Piano Keyboard

Una vez efectuadas las conexiones, se deben de conectar los microcontroladores a un PC con Code Composer Studio y escribir el programa correspondiente en la memoria de los mismos.

Hecho esto, el controlador MIDI comenzará a funcionar. Primero, se muestra en la interfaz una primera pantalla de presentación que se mantendrá hasta que el usuario haga una pulsación. Pasada esta primera pantalla, se muestra la pantalla principal del controlador, con los distintos controles programados. Empieza en la pantalla con los controles para el canal 0. Mediante el selector que se encuentra arriba a la izquierda se puede cambiar el canal.

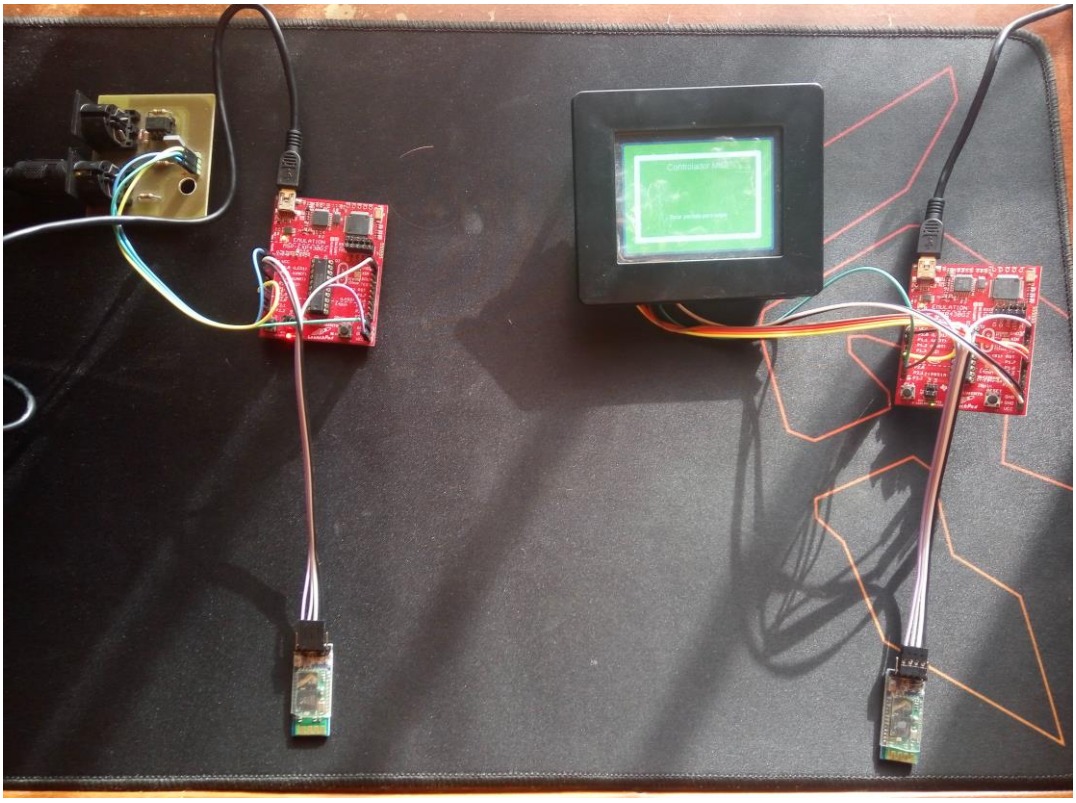

Figura 7-4. Montaje físico

Al actuar sobre alguno de los controladores, éste actualiza su posición en pantalla de manera inmediata y se envía el comando correspondiente con el nuevo valor del parámetro. Este llega al bloque receptor que lo envía al dispositivo MIDI al que se encuentra conectado.

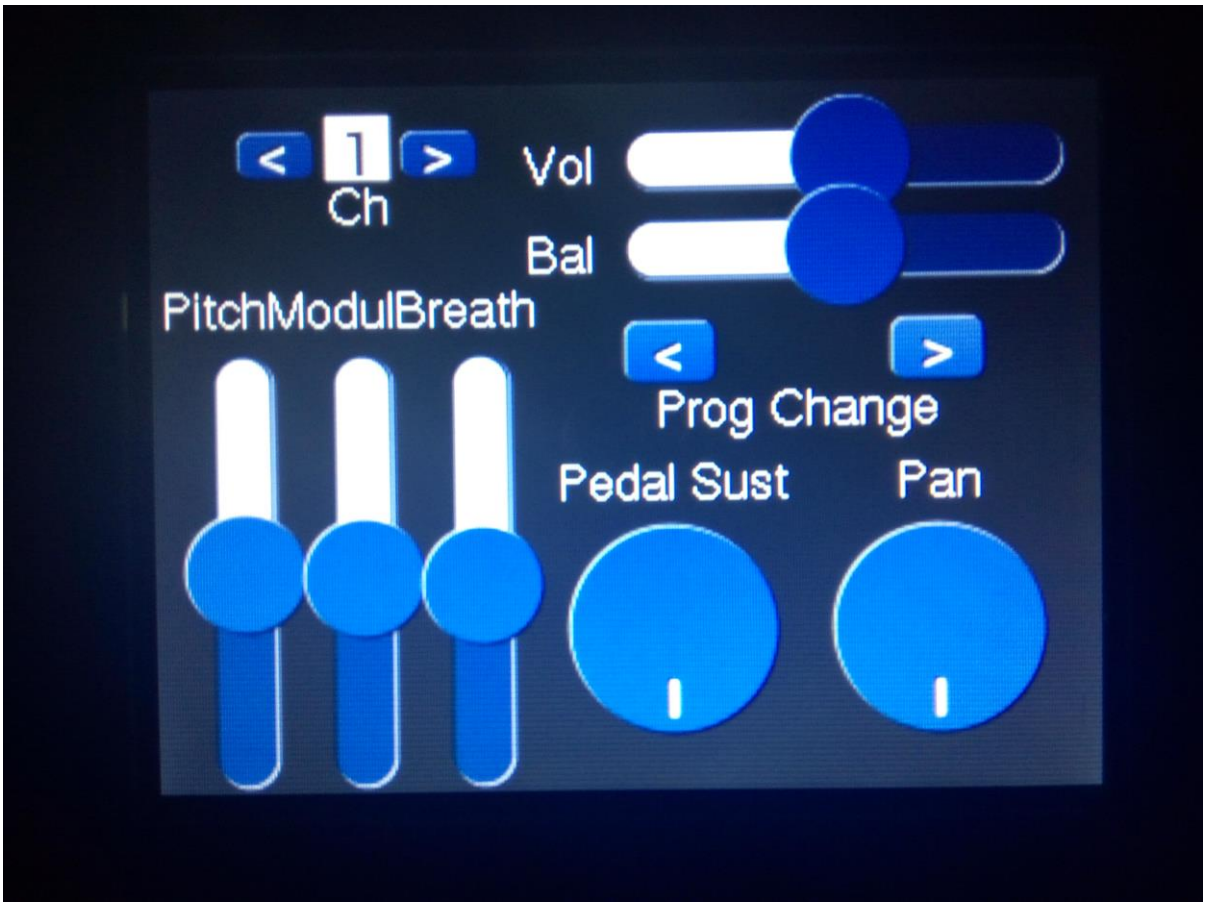

Figura 7-5. Interfaz de usuario

Mediante el programa MIDI-OX, por ejemplo, se puede observar los distintos comandos que van llegando. Si se ha configurado algún sintetizado como MIDI OUT, se podrán apreciar los cambios producidos en el sonido ante el cambio de los diferentes controles.

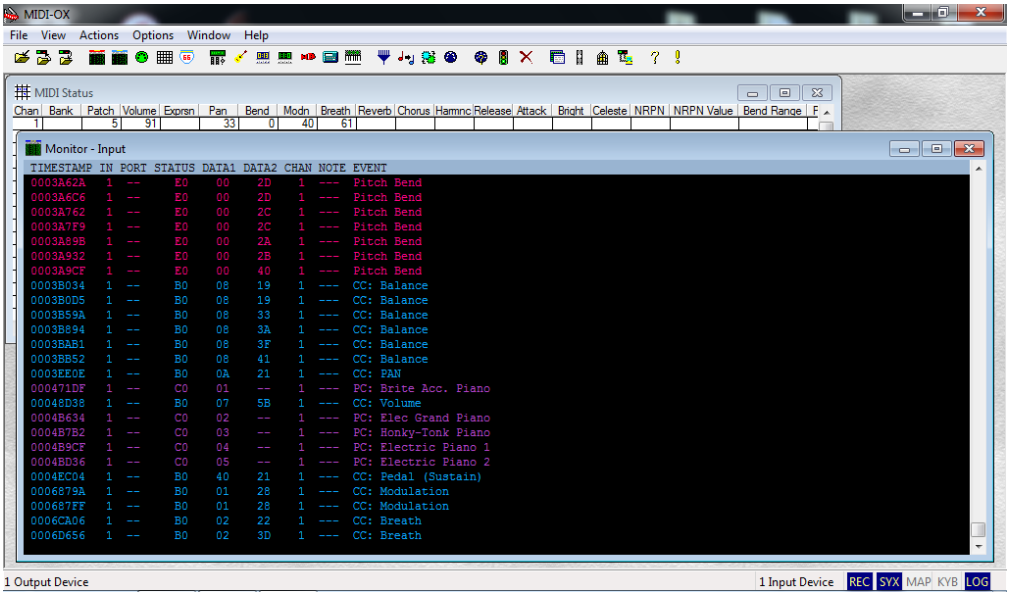

Figura 7-6. MIDI-OX y recepción de comandos

8 CONCLUSIONES Y TRABAJO FUTURO

A lo largo de este documento, se ha detallado el diseño del controlador MIDI del que trata este proyecto. Para ello, se han descrito primeramente los distintos componentes que lo conforman, haciendo un breve análisis de las distintas tecnologías utilizadas y del estado del arte de las mismas, y de las especificaciones de estos componentes. Todo ello, siempre particularizando y centrándose en aquellas más útiles y relevantes para el proyecto.

Se ha comenzado con una descripición del estándar MIDI, que es la base de este proyecto. Se han pormenorizado las características de este protocolo, tanto a nivel de software como de hardware, y se han enumerado algunos ejemplos de dispositivos MIDI que se pueden encontrar en el mercado actualmente, así como posibles alternativas al propio estándar MIDI.

Se ha proseguido con una descripción del estándar Bluetooth mediante el cual se implementa la comunicación inalámbrica en este proyecto gracias a los módulos HC-05. Se han ofrecido indicaciones para el correcto funcionamiento y configuración de los mismos.

Así mismo, se han comentado los principios básicos de las pantallas táctiles actuales y la relevancia de su aparición en el mercado. En especial, se ha descrito con profundidad la pantalla VM800 que se utiliza como interfaz del proyecto para entender el funcionamiento de la misma.

Como dispositivo clave del proyecto, se han abordado brevemente, el mercado de microcontroladores actuales, enumerando algunas de las características más importantes que se han de considerar en la elección de los mismos. Se ha particularizado, entonces, para el microcontrolador MSP430G2553 de Texas Instruments que se utiliza en el proyecto. Se han expuesto sus especificaciones, sus periféricos y se ha profundizado en aquellos que son utilizados para implementar el controlador, viendo todos los registros que son necesarios conocer para programar correctamente el funcionamiento de estos periféricos. En particular, se ha focalizado la atención, sobre todo, en los módulos de comunicación del microcontrolador, que son esenciales para este proyecto.

Tras plantear y desarrollar todos estos conceptos, indispensables para llevar a cabo el proyecto, se ha descrito en profundidad la solución implementada a nivel de código y de conexionado físico de los elementos.

Finalmente, todo lo expuesto a lo largo de este documento, se lleva a la práctica y se comprueba el funcionamiento del controlador MIDI. De esta manera, se cumplen los objetivos del proyecto, que era diseñar un dispositivo MIDI que permitiese controlar a distancia otro dispositivo MIDI distinto. Este hecho, va en línea con la tendencia actual de la aparición cada vez mayor de productos electrónicos que prescinden de conexiones con cables y eligen comunicaciones inalámbricas para mayor comodidad del usuario. Esto puede ser especialmente útil a la hora de llevar a cabo una grabación con numerosos instrumentos y dispositivos, donde se puede apreciar la libertad de movimiento que permite un dispositivo con estas características. En particular, se utiliza la tecnología Bluetooth, cuya presencia en el mercado actual es muy notable, con cada vez más productos que ofrecen esta conectividad para atraer un mayor número de compradores.

Así mismo, se ha desarrollado un producto que al seguir el estándar MIDI, es compatible con un número elevadísimo de dispositivos, prácticamente cualquier instrumento digital que se pueda adquirir o encontrar en un estudio de grabación tanto en el presente como en años futuros. Esto se debe al altísimo grado de implantación del estándar MIDI en esta industria, que pese a la aparición de otros estándares competidores más avanzados, no se prevee que cambie fácilmente.

Adicionalmente, y de nuevo siguiendo tendencias actuales del mercado, se trata de un producto que posee una interfaz táctil, acorde a las preferencias actuales de los consumidores, que apuestan cada vez más por dispositivos que ofrezan una interacción sencilla, accesible y agradable.

Por último, se ha podido profundizar en la familia de microcontroladores MSP430 de Texas Instruments. Un fabricante de sobras reconocido en el sector, con una fuerte implantación de sus productos, y una familia de microcontroladores que sigue actualizándose, y con una fuerte compatibilidad entre los distintos micros disponibles. Esto es siempre algo positivo de cara a futuras ampliaciones.

Como posibles fuentes de trabajo futuro y de mejoras se pueden citar:

- A modo de demostración tecnológica, el controlador se ha desarrollado con un cableado bastante simple, lo cual en ningún caso sería lo suficientemente fiable para un producto con vistas de comercialización. Sería necesario, en ese caso, mejorar este apartado soldando estos cables permanentemente o desarrollando una placa de conexión que fuera lo suficientemente fiable y reducida. En este sentido, sería también necesario dotar al sistema de sendas baterías para un funcionamiento verdaderamente independiente, así como de una envoltura adecuada, lo más reducida posible que proteja el producto de agentes externos, dejando tan solo la pantalla y los conectores del receptor al descubierto.
- Ampliación del número disponible de parámetros a controlar, y de los canales disponibles. El código del programa del microcontrolador emisor está cerca del límite de la capacidad de memoria del microcontrolador utilizado. Esto limita el número de pantallas y de elementos que se pueden diseñar. Una sustitución por un microcontrolador con mayor capacidad, permitiría ampliar el código y ofrecer una mayor cantidad de parámetros a controlar. En general, mejoras a nivel de hardware que aumenten las prestaciones del dispositivo. Éstas, sin embargo, tendrán un impacto negativo en el coste que hay que ponderar.

REFERENCIAS

[1] MIDI Association. “MIDI Tutorials”. Disponible en: https://www.midi.org/articles/tutorials. Fecha de consulta: Agosto de 2016.

[2] Thomann GmbH. Tienda online de dispositivos musicales. Disponible en: https://www.thomann.de/es/index.html. Fecha de consulta: Agosto de 2016.

[3] Gearmusic Ltd. Tienda online de dispositivos musicales. Disponible en: http://www.gear4music.es/es/. Fecha de consulta: Agosto de 2016.

[4] Manuel Ángel Perales Esteve. “Apuntes Electrónica de Consumo”. Universidad de Sevilla.

[5] Athan Billias. Artículo en MIDI Association. “MIDI History:Chapter 6-MIDI Is Born 1980-1983”. Disponible en: https://www.midi.org/articles/midi-history-chapter-6-midi-is-born-1980-1983. Fecha de consulta: Agosto de 2016.

[6] Manuel Ángel Perales Esteve. “Apuntes Equipos de Audio, Video y TV” Parte II (Equipos y sistemas de audio digital). Universidad de Sevilla.

[7] Open Sound Control. “Introduction to OSC”. Disponible en: http://opensoundcontrol.org/introduction-osc. Fecha de consulta: Agosto de 2016.

[8] Wikipedia. “Bluetooth”. Disponible en: https://es.wikipedia.org/wiki/Bluetooth. Fecha de consulta: Agosto de 2016.

[9] Bluetooth SIG, Inc. “Bluetooth Low Energy”. Disponible en: https://www.bluetooth.com/what-is-bluetooth-technology/bluetooth-technology-basics/low-energy. Fecha de consulta: Agosto de 2016.

[10] Bluetooth SIG, Inc. “Bluetooth® 5 quadruples range, doubles speed, increases data broadcasting capacity by 800%”. Disponible en: https://www.bluetooth.com/news/pressreleases/2016/06/16/-bluetooth5-quadruples-rangedoubles-speedincreases-data-broadcasting-capacity-by-800. Fecha de consulta: Agosto de 2016.

[11] Guangzhou HC Information Technology Co., Ltd. “HC Serial Bluetooth Products User Instructional Manual”. Disponible en: https://www.rcscomponents.kiev.ua/datasheets/hc_hc-05-user-instructions-bluetooth.pdf. Fecha de consulta: Agosto de 2016.

[12] “HC-03/05 Embedded Bluetooth Serial Communication Module AT command set”. Disponible en: http://www.linotux.ch/arduino/HC-0305_serial_module_AT_commamd_set_201104_revised.pdf. Fecha de consulta: Agosto de 2016.

[13] Future Technology Devices International Ltd. “FT800 Embedded Video Engine Datasheet Version 1.1”. Disponible en: http://www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT800.pdf. Fecha de consulta: Agosto de 2016.

[14] Future Technology Devices International Ltd. “FT800 Example with 8-bit MCU”. Disponible en: http://www.ftdichip.com/Support/Documents/AppNotes/AN_259%20FT800%20Example%20with%208-bit%20MCU.pdf. Fecha de consulta: Agosto de 2016.

[15] Future Technology Devices International Ltd. “FT800 Programmer Guide Version 0.8”. Disponible en: http://www.ftdichip.com/Support/Documents/ProgramGuides/FT800%20Programmers%20Guide.pdf Fecha de consulta: Agosto de 2016.

[16] Wikipedia. “Microcontrolador”. Disponible en: https://es.wikipedia.org/wiki/Microcontrolador#Memoria. Fecha de consulta: Agosto de 2016.

[17] Texas Instruments. “Mixed Signal Controller MSP430G2x53, MSP430G2x13 Datasheet”. Disponible en: http://www.ti.com/lit/ds/symlink/msp430g2253.pdf. Fecha de consulta: Agosto de 2016.

[18] Crash-Bang Prototyping. “Getting Started with MSP430 Timers”. Disponible en: http://www.crash-bang.com/getting-started-msp430-timers-2/. Fecha de consulta: Agosto de 2016.

[19] Texas Instruments. “MSP430x2xx Family User's Guide”. Disponible en: http://www.ti.com/lit/ug/slau144j/slau144j.pdf. Fecha de consulta: Agosto de 2016.

[20] Texas Instruments. “Implementing a UART Function With TimerA3”. Disponible en: http://www.ti.com/lit/an/slaa078a/slaa078a.pdf. Fecha de consulta: Agosto de 2016.

GLOSARIO

ANEXO I

En este anexo se adjunta el código para el microcontrolador del bloque receptor:

```
#include <msp430.h>

#define UART_TBIT           (8000000 / 31250)   // Bit time, para ir actualizando el
timer de la UART SW.

unsigned char mensaje1; // Mensaje MIDI
unsigned int txData;    // Variable interna para transmision SW UART
void TimerA_UART_tx(unsigned int byte);  // Declaracion funcion transmision SW UART
int i=0, FinRx=0;       // Variables auxiliares

int main(void) {

      //Configuracion HW UART-> Puerto 2: (TX BIT2 RX BIT1)
      //Conexion MSP430-HC05

    WDTCTL = WDTPW | WDTHOLD;    // Parar Timer del Watchdog
    if (CALBC1_8MHZ==0xFF)       // Por si hay error con la calibración del reloj
        {
          while(1);
        }

    DCOCTL = 0;
    BCSCTL1 = CALBC1_8MHZ;
    DCOCTL = CALDCO_8MHZ;     // Configuracion del oscilador interno a 8 MHz

    P1SEL|= BIT1+BIT2;        // Seleccionamos funciones de los pines 1.1 y 1.2 (UART)
    P1SEL2|= BIT1+BIT2;       // P1.1 = RXD (Recibir), P1.2=TXD (Transmitir)

    UCA0CTL1 |= UCSWRST;      // Reset UART previo a configuracion
    UCA0CTL1 |= UCSSEL_2;     // Elegimos reloj SMCLK. UCSSEL es "10" ya que UCSEEL_2
es 0x80

    UCA0BR0 = 0x45;           // Configuracion velocidad UART Hardware
    UCA0BR1 = 0x0;            // 115200 bps
    UCA0MCTL = 0xAA;

    UCA0CTL1 &= ~UCSWRST;     // Desactivar Reset

    P1DIR=BIT0+BIT6;          //Pines 1.0 y 1.6 como salida (LEDs)
    P1REN=BIT3;               //Habilitar Resistencia (1.3 es entrada por defecto)
    P1OUT = BIT3;             //Resistencia Pull-up (1.3 es el boton del Launchpad)

    IE2 |= UCA0RXIE;          // Activo Interrupción para la recepción de datos
```

```
    //Configuracion SW UART-> Puerto 2: (TX BIT1(0) RX BIT2(3))
    //Conexion Disp.MIDI-MSP430

    P2OUT = 0x00;                               // Inicializo valores GPIO
    P2SEL|= BIT0 + BIT3;                        // Los pines los configuro como Timer A
    P2DIR = 0xFF & ~BIT3;                 // Salvo RX, pines como salida

    TA1CCTL0 = OUT;                       // En reposo, la linea a 1 (salida del Timer1)
    TA1CCTL1 = SCS + CM1 + CAP + CCIE; // Captura síncrona, Flanco de Bajada, Modo
Captura, Habilita Interrupcion
    TA1CTL = TASSEL_2 + MC_2;             // Reloj SMCLK, Modo Continuo

    P1OUT|=BIT0;    // P1.0 encendido
    P1OUT&=~BIT6;   // P1.6 apagado
    __bis_SR_register(GIE);               // Habilitar máscara global de interrupciones

    while(1)
    {
       if(!(P1IN&BIT3))
       {
             P1OUT|=BIT6; // Cuando se pulsa el boton, se transmite mensaje de prueba
             P1OUT&=~BIT0;                // Cambio iluminacion LEDs
             mensaje1=0x90;
             TimerA_UART_tx(mensaje1);  // Transmito Status byte
             __delay_cycles(10);
             mensaje1=0x38;
             TimerA_UART_tx(mensaje1);  //Transmito Data byte 1
             __delay_cycles(10);
             mensaje1=0x64;
             TimerA_UART_tx(mensaje1);  //Transmito Data byte 2
             __delay_cycles(10000);
        }
        else
        {                       // Boton no pulsado, funcionamiento normal
             P1OUT|=BIT0;      // P1.0 encendido
             P1OUT&=~BIT6;     // P1.6 apagado
             if(FinRx)
             {
                   FinRx=0;   // Si la recepción se ha completado con exito, se
                   envia el dato
                   TimerA_UART_tx(mensaje1); // El dato se envia por la SW UART
             }
        }
    }
}

// Interrupcion HW UART. Lo que llega por el canal RX (modulo HC05), lo guardo en
memoria

#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR(void)
{
      P1OUT|=BIT6;         // P1.6 encendido
      P1OUT&=~BIT0;        // P1.0 apagado
      mensaje1=UCA0RXBUF;
      FinRx=1;             // Indico recepcion de dato
}
```

```
// Funcion de transmision para SW UART

void TimerA_UART_tx(unsigned int byte)
{
    while (TA1CCTL0 & CCIE);           // Aseguramos que podemos transmitir
    TA1CCR0 = TAR;                     // Estado actual del timer, se carga en el
registro CCR0
    TA1CCR0 += UART_TBIT;              // Sumamos bit time al registro CCR0, para
mantener el baudrate
    TA1CCTL0 = OUTMOD0 + CCIE;         // Activar interrupcion Timer. Modo OUTMODx en
SET (línea a 1)
    txData = byte;                     // Escribimos variable txData para transmitir
(8 bits + start y stop)
    txData |= 0x100;                   // Bit de Stop (lo ponemos en MSB)
    txData <<= 1;                      // Bit de Start (Trasladar byte a la izquierda
para poner un 0 como LSB)
}

// Interrupcion del Timer para SW UART

#pragma vector = TIMER1_A0_VECTOR
__interrupt void Timer_A0_ISR(void)
{
    static unsigned char txBitCnt = 10;      //Numero de bits que quedan por
transmitir

    TA1CCR0 += UART_TBIT;                    // Sumamos bit time
    if (txBitCnt == 0)
    {                                        // Si todos los bits transmitidos
        TA1CCTL0 &= ~CCIE;                   // Desactivamos interrupcion
        txBitCnt = 10;                       // Resetear contador de bits a 10.
    }
    else
    {
        if (txData & 0x01)
        {                                    // Si bit actual es "1"
          TA1CCTL0 &= ~OUTMOD2;              // TX '1' -> Mantener OUTMOD0 (Set a 1)
        }
        else
       {                                 // Si bit actual es "0"
          TA1CCTL0 |= OUTMOD2;                  // TX '0' -> Escribir un OUTMOD2, es
equivalente a un Reset, porque el bit OUTMOD0 permanece (Linea pasa de 1 a 0)
        }
        txData >>= 1;                        //Movemos bit a la derecha cuando LSB
ha sido transmitido
        txBitCnt--;                          //Restamos 1 en contador de bits
    }
}
```

ANEXO II

En este anexo se adjunta el código para el microcontrolador del bloque emisor. Empezando por el main:

```
#include "FT800.h"
#include "HC05.h"
#include <msp430.h>
#include <math.h>

#define dword long
#define byte char

unsigned char Vp1=64, Vp2=64, Vp3=64, Vp4=64; //Inicializo Volumen
unsigned char Bp1=64, Bp2=64, Bp3=64, Bp4=64; //Inicializo Balance
unsigned char Pp1=64, Pp2=64, Pp3=64, Pp4=64; //Inicializo Pitch (MSB)
unsigned char Mp1=0, Mp2=0, Mp3=0, Mp4=0;     //Inicializo Modulation
unsigned char Brp1=0, Brp2=0, Brp3=0, Brp4=0; //Inicializo Breath
unsigned char Enc1a=0, Enc1b=0, Enc2a=0, Enc2b=0, Enc3a=0, Enc3b=0, Enc4a=0, Enc4b=0;
//Inicializo Encoders
unsigned int Th1a=0x0000, Th1b=0x0000, Th2a=0x0000, Th2b=0x0000, Th3a=0x0000,
Th3b=0x0000, Th4a=0x0000, Th4b=0x0000;   //Inicializo valores de theta
int X1a=0x0000, X1b=0x0000, X2a=0x0000, X2b=0x0000, X3a=0x0000, X3b=0x0000,
X4a=0x0000, X4b=0x0000;
int Y1a=0x0000, Y1b=0x0000, Y2a=0x0000, Y2b=0x0000, Y3a=0x0000, Y3b=0x0000,
Y4a=0x0000, Y4b=0x0000;

char chipid = 0;                        // Valor registro ChipID

extern unsigned long POSX, POSY, BufferXY;
extern unsigned int CMD_Offset;
unsigned long REG_TT[6];
const unsigned long REG_CAL[6]={21959,177,4294145463,14,4294950369,16094853};
char Prog[60];

unsigned char comando;
char j=0, FinRx=0;

void main(void)
{
      int i;
      int Channel=1;
      int PitchPULS1=0,PitchPULS2=0,PitchPULS3=0,PitchPULS4=0;
      unsigned int Program1=0,Program2=0,Program3=0,Program4=0;

// Configuracion e Inicializacion de los perifericos

    HAL_Configure_MCU();  // Configuracion uC
    P2DIR|=BIT3+BIT5;
    InicializaHC05();     // Configuracion UART
    Inicia_pantalla();    // Inicializacion de pantalla
    espera(500);          // Esperar 500ms
```

```
// Pantalla incial de presentacion

      Nueva_pantalla(0x101010);   //Color de fondo codificado como numero de 24
bits. 10=R 10=G 10=B
      ComColor(0,105,117);        //Paso RGB del siguiente objeto que voy a pintar:
      Comando(CMD_LINEWIDTH+80); //Ancho de la línea es tal que 80=16*ancho
      Comando(CMD_BEGIN_RECTS);  //Rectangulos: Todos los puntos que pase ahora
seran esquinas de rectangulos
      ComVertex2ff(15,15);       //1er vertice
      ComVertex2ff(305,225);    //2do vertice
      ComColor(65,202,42);      //Cambio color
      ComVertex2ff(20,20);
      ComVertex2ff(300,220);    //Termino marco
      Comando(CMD_END);         // Proximos vertices no seran de rectangulos

      ComColor(0xff,0xff,0xff);  //Blanco
      ComTXT(160,50, 22, OPT_CENTERX,"Controlador MIDI");
      ComTXT(160,150, 20, OPT_CENTERX," Tocar pantalla para seguir"); //Texto
      Comando(CMD_BEGIN_LINES);  //Voy a hacer rectangulos pintando lineas
      Comando(CMD_LINEWIDTH+80);
      ComVertex2ff(40,40); //Cada pareja de vertices definen una linea
      ComVertex2ff(280,40);
      ComVertex2ff(280,40);
      ComVertex2ff(280,200);
      ComVertex2ff(280,200);
      ComVertex2ff(40,200);
      ComVertex2ff(40,200);
      ComVertex2ff(40,40);
      Comando(CMD_END);

      Dibuja();      // Ordeno ejecución lista de comandos. Dibujo pantalla inicial
      Espera_pant(); // Se mantiene pantalla actual hasta pulsacion

    for(i=0;i<6;i++)Esc_Reg(REG_TOUCH_TRANSFORM_A+4*i, REG_CAL[i]); //Calibracion

// Tras pulsar entramos en bucle principal con la interfaz del controlador

    while(1)
    {
     if(!(P1IN&BIT3))
     { //si P1.3=0 (Boton pulsado) ENVIO DE COMANDO DE PRUEBA
       comando=0x90;
       UCA0TXBUF=comando;  //Transmito Status Byte
       __delay_cycles(100000);
       comando=0x38;
       UCA0TXBUF=comando;  //Transmito Data Byte 1
       __delay_cycles(100000);
       comando=0x64;
       UCA0TXBUF=comando;  //Transmito Data Byte 2
       __delay_cycles(1000000);
     }
    else
     {
       Lee_pantalla(); // Leo la pantalla. Resultado en vbles globales POSX POSY

       Nueva_pantalla(0x101010); // Comienzo diseno nueva pantalla
```

```
ComColor(0xff,0xff,0xff);
Comando(CMD_BEGIN_RECTS);              // Recuadro Canal
ComVertex2ff(60,10);                   // 1er Vertice
ComVertex2ff(80,30);                   // 2do Vertice
ComTXT(70,30, 22, OPT_CENTERX,"Ch");   // Texto
ComTXT(138,17, 22, OPT_CENTERX,"Vol");
ComTXT(138,47, 22, OPT_CENTERX,"Bal");

ComTXT(20,65, 22, OPT_CENTERX,"Pitch");
ComTXT(60,65, 22, OPT_CENTERX,"Modul");
ComTXT(105,65, 22, OPT_CENTERX,"Breath");

ComTXT(175,125, 22, OPT_CENTERX,"Pedal Sust");
ComTXT(265,125, 22, OPT_CENTERX,"Pan");
ComTXT(217,100, 22, OPT_CENTERX,"Prog Change");
Comando(CMD_END);


//-------------Actualizo Channel

if(POSX>30 && POSX<55 && POSY>15 && POSY<30) //Channel << (-1)
{
    ComButton(30,15,25,15,28,256,"<");    // Pinto boton pulsado en esa zona
    Channel--; if (Channel<=1) Channel=1; // Channel min = 1
    espera(100);
}
else // Si no he pulsado no actualizo Channel
{
    ComButton(30,15,25,15,28,0,"<");      // Pinto boton con efecto no pulsado
}

if(POSX>85 && POSX<110 && POSY>15 && POSY<30) //Channel >> (+1)
{
    ComButton(85,15,25,15,28,256,">");
    Channel++; if (Channel>=4) Channel=4;  //Channel max = 4
    espera(100);
}
else
{
    ComButton(85,15,25,15,28,0,">");
}


//-------------Pinto controles en pantalla

if (Channel==1) // Pantalla canal 1
{
    if(POSX>170 && POSX<297 && POSY>15 && POSY<35)
    {        // Pulso en el slider de Volumen
             Vp1=POSX-170; // Actualizo posicion del slider (0-127)
             ComSlider(170,15,127,20,256,Vp1,127); // CMD_SLIDER
             comando=0xB0; // CC-Channel 1
             UCA0TXBUF=comando;
              __delay_cycles(10000);
             comando=0x07; // CC7
             UCA0TXBUF=comando;
              __delay_cycles(10000);
             comando=Vp1;  // Valor volumen (0vvvvvvv)
```

```
                UCA0TXBUF=comando;
                 __delay_cycles(1000000);
              }            // Dibujo el slider pulsado (sin efecto 3D)
              else
              {ComSlider(170,15,127,20,0,Vp1,127);}  // Slider sin pulsar (efecto 3D)

              if(POSX>170 && POSX<297 && POSY>45 && POSY<65)
              {      // Pulso en el slider de Balance
                     Bp1=POSX-170; // Actualizo posicion del slider (0-127)
                     ComSlider(170,45,127,20,256,Bp1,127); // CMD_SLIDER
                     comando=0xB0; // CC-Channel 1
                     UCA0TXBUF=comando;
                     __delay_cycles(10000);
                     comando=0x08; // CC8
                     UCA0TXBUF=comando;
                     __delay_cycles(10000);
                     comando=Bp1;  // Valor balance (0bbbbbbb)
                     UCA0TXBUF=comando;
                     __delay_cycles(1000000);
                     }            //Dibujo el slider pulsado (sin efecto 3D)
              else
              {ComSlider(170,45,127,20,0,Bp1,127);}  // Slider sin pulsar  (efecto 3D)

              if(POSX>20 && POSX<40 && POSY>100 && POSY<227)
              {     // Pulso en el slider de Pitch Bend. EL SLIDER DE PITCH VUELVE A
POSICION ORIGINAL TRAS DEJAR DE PULSAR
                     Pp1=POSY-100; // Actualizo posicion del slider (0-127)
                     ComSlider(20,100,20,127,256,Pp1,127); // CMD_SLIDER
                     PitchPULS1=1; // Vble auxiliar indica pulsación del slider
                     comando=0xE0; // Pitch-Channel 1
                     UCA0TXBUF=comando;
                     __delay_cycles(10000);
                     comando=0x00; // LSB byte (0x00)
                     UCA0TXBUF=comando;
                     __delay_cycles(10000);
                     comando=Pp1;  // MSB Byte - Valor pitch (0ppppppp)
                     UCA0TXBUF=comando;
                     __delay_cycles(1000000);
              }            //Dibujo el slider pulsado
              else
              {
                     ComSlider(20,100,20,127,0,Pp1,127);
                      if(PitchPULS1==1)
                            {
                                   PitchPULS1=0; Pp1=64;    //Al dejar de pulsar en el
                            control de Pitch vuelve
                                   comando=0xE0; // Pitch-Channel 1
                                   UCA0TXBUF=comando;
                                   __delay_cycles(10000);
                                   comando=0x00; // LSB byte (0x00)
                                   UCA0TXBUF=comando;
                                   __delay_cycles(10000);
                                   comando=Pp1;  // MSB Byte - Valor pitch
                                   UCA0TXBUF=comando;
                                   __delay_cycles(1000000);
                            }
               }     // Slider sin pulsar  (efecto 3D)
```

```
if(POSX>60 && POSX<80 && POSY>100 && POSY<227)
{      // Pulso en el slider de Modulation
       Mp1=POSY-100; // Actualizo posicion del slider (0-127)
       ComSlider(60,100,20,127,256,Mp1,127); // CMD_SLIDER
       comando=0xB0; // CC-Channel 1
       UCA0TXBUF=comando;
       __delay_cycles(10000);
       comando=0x01; // CC1
       UCA0TXBUF=comando;
       __delay_cycles(10000);
       comando=Mp1;  // Valor Modulation (0bbbbbbb)
       UCA0TXBUF=comando;
       __delay_cycles(1000000);
}             //Dibujo el slider pulsado
else
{ComSlider(60,100,20,127,0,Mp1,127);}  // Slider sin pulsar  (efecto 3D)

if(POSX>100 && POSX<120 && POSY>100 && POSY<227)
{      // Pulso en el slider de Breath
       Brp1=POSY-100; // Actualizo posicion del slider (0-127)
       ComSlider(100,100,20,127,256,Brp1,127); // CMD_SLIDER
       comando=0xB0; // CC-Channel 1
       UCA0TXBUF=comando;
       __delay_cycles(10000);
       comando=0x02; // CC2
       UCA0TXBUF=comando;
       __delay_cycles(10000);
       comando=Brp1; // Valor Breath (0bbbbbbb)
       UCA0TXBUF=comando;
       __delay_cycles(1000000);
}             // Dibujo el slider pulsado
else
{ComSlider(100,100,20,127,0,Brp1,127);}      // Slider sin pulsar

if(POSX>140 && POSX<210 && POSY>150 && POSY<220)
{      // Pulso en el encoder 1a (Pedal Sustain)
       X1a=POSX-175;Y1a=POSY-185; // Lectura en ejes X,Y
       if(abs(X1a)>=12||abs(Y1a)>=12)
       {
              Th1a=Theta(X1a,Y1a);    //Calculo angulo (posicion encoder)
              Enc1a=Encoder(Th1a);    //Obtengo valor midi
       }
       ComDial(175,185,35,256,Th1a); // CMD_DIAL
       comando=0xB0; //  CC-Channel 1
       UCA0TXBUF=comando;
       __delay_cycles(10000);
       comando=0x40; //  CC64
       UCA0TXBUF=comando;
       __delay_cycles(10000);
       comando=Enc1a;// Valor Pedal (0bbbbbbb)
       UCA0TXBUF=comando;
       __delay_cycles(1000000);
}             //Dibujo el encoder pulsado (sin efecto 3D)
else
{ComDial(175,185,35,0,Th1a);}   //Encoder sin pulsar  (con efecto 3D)
```

```
        if(POSX>230 && POSX<300 && POSY>150 && POSY<220)
        {       // Pulso en el encoder 1b (Pan)
                X1b=POSX-265;Y1b=POSY-185;
                if(abs(X1b)>=12||abs(Y1b)>=12)
                {
                        Th1b=Theta(X1b,Y1b);            //Calculo angulo
                        Enc1b=Encoder(Th1b);            //Obtengo valor midi
                }
                ComDial(265,185,35,256,Th1b); // CMD_SLIDER
                comando=0xB0; // CC-Channel 1
                UCA0TXBUF=comando;
                __delay_cycles(10000);
                comando=0xA; // CC10
                UCA0TXBUF=comando;
                __delay_cycles(10000);
                comando=Enc1b;// Valor Pan (0bbbbbbb)
                UCA0TXBUF=comando;
                __delay_cycles(1000000);
        }               // Dibujo el encoder pulsado (sin efecto 3D)
        else
        {ComDial(265,185,35,0,Th1b);}   // Encoder sin pulsar  (con efecto 3D)

        if(POSX>160 && POSX<190 && POSY>80 && POSY<100) //Programa<< (-1)
        {
                ComButton(160,80,30,20,28,256,"<");      //Pinto boton pulsado
                Program1--; if (Program1<=0) Program1=0; //Program1 min = 0
                comando=0xC0;     // Program Change-Channel 1
                UCA0TXBUF=comando;
                __delay_cycles(10000);
                comando=Program1; // Data1
                UCA0TXBUF=comando;
                __delay_cycles(1000000);
        }
        else
        {ComButton(160,80,30,20,28,0,"<");}

        if(POSX>250 && POSX<280 && POSY>80 && POSY<100) //Programa>> (+1)
        {
                ComButton(250,80,30,20,28,256,">");
                Program1++; if (Program1>=127) Program1=127; //Program1 max = 127
                comando=0xC0;     // Program Change-Channel 1
                UCA0TXBUF=comando;
                __delay_cycles(10000);
                comando=Program1; // Data1
                UCA0TXBUF=comando;
                __delay_cycles(1000000);
        }
        else
        {ComButton(250,80,30,20,28,0,">");}

        ComColor(0x10,0x10,0x10);
        ComTXT(70,10, 28, OPT_CENTERX,"1"); //Pintar numero canal en el recuadro
        ComColor(0xff,0xff,0xff);  //Blanco
}
```

```
if (Channel==2)
{
	if(POSX>170 && POSX<297 && POSY>15 && POSY<35)
	{	// Pulso en el slider de Volumen
		Vp2=POSX-170;
		ComSlider(170,15,127,20,256,Vp2,127);
		comando=0xB1; // CC-Channel 2
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x07; // CC7
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Vp2;  // Valor volumen (0vvvvvvv)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el slider pulsado
	else
	{ComSlider(170,15,127,20,0,Vp2,127);}		// Slider sin pulsar

	if(POSX>170 && POSX<297 && POSY>45 && POSY<65)
	{	// Pulso en el slider de Balance
		Bp2=POSX-170;
		ComSlider(170,45,127,20,256,Bp2,127);
		comando=0xB1; // CC-Channel 2
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x08; // CC8
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Bp2;  // Valor balance (0bbbbbbb)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
		}		// Dibujo el slider pulsado (sin efecto 3D)
	else
	{ComSlider(170,45,127,20,0,Bp2,127);}  // Slider sin pulsar

	if(POSX>20 && POSX<40 && POSY>100 && POSY<227)
	{	// Pulso en el slider de Pitch Bend
		Pp2=POSY-100;
		ComSlider(20,100,20,127,256,Pp2,127);
		PitchPULS2=1;
		comando=0xE1; // Pitch-Channel 2
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x00; // LSB byte (0x00)
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Pp2;  // MSB Byte - Valor Pitch
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		//Dibujo el slider pulsado
	else
	{
		ComSlider(20,100,20,127,0,Pp2,127);
		if(PitchPULS2==1)
		{
			PitchPULS2=0; Pp2=64;	//Al dejar de pulsar en el control
		de Pitch vuelve
			comando=0xE1; //--Pitch-Channel 2
```

```
                        UCA0TXBUF=comando;
                        __delay_cycles(10000);
                        comando=0x00; // LSB byte (0x00)
                        UCA0TXBUF=comando;
                        __delay_cycles(10000);
                        comando=Pp2; // MSB Byte - Valor Pitch
                        UCA0TXBUF=comando;
                        __delay_cycles(1000000);
                  }
            }  //Slider sin pulsar

      if(POSX>60 && POSX<80 && POSY>100 && POSY<227)
      {     //Pulso en el slider de Modulation
            Mp2=POSY-100;
            ComSlider(60,100,20,127,256,Mp2,127);
            comando=0xB1; // CC-Channel 2
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=0x01; // CC1
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=Mp2;  // Valor Modulation (0bbbbbbb)
            UCA0TXBUF=comando;
            __delay_cycles(1000000);
      }           //Dibujo el slider pulsado
      else
      {ComSlider(60,100,20,127,0,Mp2,127);}        // Slider sin pulsar

      if(POSX>100 && POSX<120 && POSY>100 && POSY<227)
      {     // Pulso en el slider de Breath
            Brp2=POSY-100;
            ComSlider(100,100,20,127,256,Brp2,127);
            comando=0xB1; // CC-Channel 2
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=0x02; // CC2
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=Brp2; // Valor Breath (0bbbbbbb)
            UCA0TXBUF=comando;
            __delay_cycles(1000000);}       //Dibujo el slider pulsado
      else
      {ComSlider(100,100,20,127,0,Brp2,127);}      //Slider sin pulsar

      if(POSX>140 && POSX<210 && POSY>150 && POSY<220)
      {     // Pulso en el encoder 2a (Pedal Sustain)
            X2a=POSX-175;Y2a=POSY-185;
            if(abs(X2a)>=12||abs(Y2a)>=12)
            {
                  Th2a=Theta(X2a,Y2a);            // Calculo angulo
                  Enc2a=Encoder(Th2a);            // Obtengo valor midi
            }
            ComDial(175,185,35,256,Th2a);
            comando=0xB1; // CC-Channel 2
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=0x40; // CC64
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=Enc2a; // Valor Pedal (0bbbbbbb)
```

```
            UCA0TXBUF=comando;
            __delay_cycles(1000000);
        }           // Dibujo el encoder pulsado
        else
        {ComDial(175,185,35,0,Th2a);}   // Encoder sin pulsar  (con efecto 3D)

        if(POSX>230 && POSX<300 && POSY>150 && POSY<220)
        {       // Pulso en el encoder 2b (Pan)
            X2b=POSX-265;Y2b=POSY-185;
            if(abs(X2b)>=12||abs(Y2b)>=12)
            {
                Th2b=Theta(X2b,Y2b);            // Calculo angulo
                Enc1b=Encoder(Th2b);            // Obtengo valor midi
            }
            ComDial(265,185,35,256,Th2b);
            comando=0xB1; // CC-Channel 2
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=0xA;  // CC10
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=Enc2b; // Valor Pan (0bbbbbbb)
            UCA0TXBUF=comando;
            __delay_cycles(1000000);
        }           // Dibujo el encoder pulsado
        else
        {ComDial(265,185,35,0,Th2b);}   // Encoder sin pulsar  (con efecto 3D)

        if(POSX>160 && POSX<190 && POSY>80 && POSY<100) // Programa<< (-1)
        {
            ComButton(160,80,30,20,28,256,"<");    // Pinto boton pulsado
            Program2--; if (Program2<=0) Program2=0; // Program2 min = 0
            comando=0xC1; // Program Change-Channel 2
            UCA0TXBUF=comando;
            __delay_cycles(10000);
            comando=Program2; // Data1
            UCA0TXBUF=comando;
            __delay_cycles(1000000);
        }
        else
        {ComButton(160,80,30,20,28,0,"<");}

        if(POSX>250 && POSX<280 && POSY>80 && POSY<100) //Programa>> (+1)
        {
            ComButton(250,80,30,20,28,256,">");
            Program2++; if (Program2>=127) Program2=127;// Program2 max = 127
                comando=0xC1; // Program Change-Channel 2
                    UCA0TXBUF=comando;
                    __delay_cycles(10000);
                    comando=Program2; // Data1
                    UCA0TXBUF=comando;
                    __delay_cycles(1000000);}
        else
        {ComButton(250,80,30,20,28,0,">");}

            ComColor(0x10,0x10,0x10);
            ComTXT(70,10, 28, OPT_CENTERX,"2");
            ComColor(0xff,0xff,0xff);
}
```

```
if (Channel==3)
{
	if(POSX>170 && POSX<297 && POSY>15 && POSY<35)
	{	// Pulso en el slider de Volumen
		Vp3=POSX-170;
		ComSlider(170,15,127,20,256,Vp3,127);
		comando=0xB2; // CC-Channel 3
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x07; // CC7
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Vp3;  // Valor volumen (0vvvvvvv)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el slider pulsado
	else
	{ComSlider(170,15,127,20,0,Vp3,127);}  // Slider sin pulsar

	if(POSX>170 && POSX<297 && POSY>45 && POSY<65)
	{	// Pulso en el slider de Balance
		Bp3=POSX-170;
		ComSlider(170,45,127,20,256,Bp3,127);
		comando=0xB2; // CC-Channel 3
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x08; // CC8
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Bp3;  // Valor balance (0bbbbbbb)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el slider pulsado (sin efecto 3D)
	else
	{ComSlider(170,45,127,20,0,Bp3,127);}  // Slider sin pulsar

	if(POSX>20 && POSX<40 && POSY>100 && POSY<227)
	{	// Pulso en el slider de Pitch Bend
		Pp3=POSY-100;
		ComSlider(20,100,20,127,256,Pp3,127);
		PitchPULS3=1;
		comando=0xE2; // Pitch-Channel 3
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x00; // LSB byte (0x00)
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Pp3;  // MSB Byte - Valor Pitch
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el slider pulsado
	else
	{
		ComSlider(20,100,20,127,0,Pp3,127);
		if(PitchPULS3==1)
		{
			PitchPULS3=0; Pp3=64;
			comando=0xE2; // Pitch-Channel 3
			UCA0TXBUF=comando;
```

```
                __delay_cycles(10000);
                comando=0x00; // LSB byte (0x00)
                UCA0TXBUF=comando;
                __delay_cycles(10000);
                comando=Pp3; // MSB Byte - Valor Pitch
                UCA0TXBUF=comando;
                __delay_cycles(1000000);
        }
}               // Slider sin pulsar

if(POSX>60 && POSX<80 && POSY>100 && POSY<227)
{       // Pulso en el slider de Modulation
        Mp3=POSY-100;
        ComSlider(60,100,20,127,256,Mp3,127);
        comando=0xB2; // CC-Channel 3
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0x01; // CC1
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Mp3; // Valor Modulation (0bbbbbbb)
        UCA0TXBUF=comando;
        __delay_cycles(1000000);}       // Dibujo el slider pulsado
else
{ComSlider(60,100,20,127,0,Mp3,127);}  // Slider sin pulsar

if(POSX>100 && POSX<120 && POSY>100 && POSY<227)
{       // Pulso en el slider de Breath
        Brp3=POSY-100;
        ComSlider(100,100,20,127,256,Brp3,127);
        comando=0xB2; // CC-Channel 3
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0x02; // CC2
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Brp3; // Valor Breath (0bbbbbbb)
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}               // Dibujo el slider pulsado
else
{ComSlider(100,100,20,127,0,Brp3,127);}  // Slider sin pulsar

if(POSX>140 && POSX<210 && POSY>150 && POSY<220)
{       // Pulso en el encoder 3a (Pedal Sustain)
        X3a=POSX-175;Y3a=POSY-185;
        if(abs(X3a)>=12||abs(Y3a)>=12)
        {
                Th3a=Theta(X3a,Y3a);            // Calculo angulo
                Enc3a=Encoder(Th3a);            // Obtengo valor midi
        }
        ComDial(175,185,35,256,Th3a);
        comando=0xB2;  // CC-Channel 3
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0x40;  // CC64
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Enc3a; // Valor Pedal (0bbbbbbb)
        UCA0TXBUF=comando;
```

```
		__delay_cycles(1000000);
	}		// Dibujo el encoder pulsado
	else
	{ComDial(175,185,35,0,Th3a);}   //Encoder sin pulsar

	if(POSX>230 && POSX<300 && POSY>150 && POSY<220)
	{	// Pulso en el encoder 3b (Pan)
		X3b=POSX-265;Y3b=POSY-185;
		if(abs(X3b)>=12||abs(Y3b)>=12)
		{
			Th3b=Theta(X3b,Y3b);		// Calculo angulo
			Enc3b=Encoder(Th3b);		// Obtengo valor midi
		}
		ComDial(265,185,35,256,Th3b);
		comando=0xB2;  // CC-Channel 3
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0xA;   // CC10
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Enc3b; // Valor Pan (0bbbbbbb)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el encoder pulsado
	else
	{ComDial(265,185,35,0,Th3b);}   // Encoder sin pulsar

	if(POSX>160 && POSX<190 && POSY>80 && POSY<100) // Programa<< (-1)
	{
		ComButton(160,80,30,20,28,256,"<");    // Pinto boton pulsado
		Program3--; if (Program3<=0) Program3=0; // Program3 min = 0
		comando=0xC2; // Program Change-Channel 3
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Program3; // Data1
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}
	else
	{ComButton(160,80,30,20,28,0,"<");}

	if(POSX>250 && POSX<280 && POSY>80 && POSY<100) // Programa>> (+1)
	{
		ComButton(250,80,30,20,28,256,">");
		Program3++; if (Program3>=127) Program3=127;// Program3 max = 127
		comando=0xC2; // Program Change-Channel 3
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Program3; // Data1
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}
	else
	{ComButton(250,80,30,20,28,0,">");}

	ComColor(0x10,0x10,0x10);
	ComTXT(70,10, 28, OPT_CENTERX,"3");
	ComColor(0xff,0xff,0xff);
}
```

```
if (Channel==4)
{
	if(POSX>170 && POSX<297 && POSY>15 && POSY<35)
	{	// Pulso en el slider de Volumen
		Vp4=POSX-170;
		ComSlider(170,15,127,20,256,Vp4,127);
		comando=0xB3; // CC-Channel 4
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x07; // CC7
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Vp4; // Valor volumen (0vvvvvvv)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el slider pulsado
	else
	{ComSlider(170,15,127,20,0,Vp4,127);}  // Slider sin pulsar

	if(POSX>170 && POSX<297 && POSY>45 && POSY<65)
	{	// Pulso en el slider de Balance
		Bp4=POSX-170;
		ComSlider(170,45,127,20,256,Bp4,127);
		comando=0xB3; // CC-Channel 4
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x08; // CC8
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Bp4;  // Valor balance (0bbbbbbb)
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
		}		// Dibujo el slider pulsado
	else
	{ComSlider(170,45,127,20,0,Bp4,127);}  //Slider sin pulsar

	if(POSX>20 && POSX<40 && POSY>100 && POSY<227)
	{	// Pulso en el slider de Pitch Bend
		Pp4=POSY-100;
		ComSlider(20,100,20,127,256,Pp4,127);
		PitchPULS4=1;
		comando=0xE3; // Pitch-Channel 4
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=0x00; // LSB byte (0x00)
		UCA0TXBUF=comando;
		__delay_cycles(10000);
		comando=Pp4;  // MSB Byte - Valor Pitch
		UCA0TXBUF=comando;
		__delay_cycles(1000000);
	}		// Dibujo el slider pulsado
	else
	{
		ComSlider(20,100,20,127,0,Pp4,127);
		if(PitchPULS4==1)
		{
			PitchPULS4=0; Pp4=64;
			comando=0xE3; // Pitch-Channel 4
			UCA0TXBUF=comando;
```

```
                        __delay_cycles(10000);
                        comando=0x00; // LSB byte (0x00)
                        UCA0TXBUF=comando;
                        __delay_cycles(10000);
                        comando=Pp4;  // MSB Byte - Pitch
                        UCA0TXBUF=comando;
                        __delay_cycles(1000000);
                }
        }       // Slider sin pulsar

if(POSX>60 && POSX<80 && POSY>100 && POSY<227)
{       // Pulso en el slider de Modulation
        Mp4=POSY-100;
        ComSlider(60,100,20,127,256,Mp4,127);
        comando=0xB3; // CC-Channel 4
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0x01; // CC1
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Mp4;  // Valor Modulation (0bbbbbbb)
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}               // Dibujo el slider pulsado
else
{ComSlider(60,100,20,127,0,Mp4,127);}  // Slider sin pulsar

if(POSX>100 && POSX<120 && POSY>100 && POSY<227)
{       // Pulso en el slider de Breath
        Brp4=POSY-100;
        ComSlider(100,100,20,127,256,Brp4,127);
        comando=0xB3; // CC-Channel 4
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0x02; // CC2
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Brp4; // Valor Breath (0bbbbbbb)
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}               // Dibujo el slider pulsado
else
{ComSlider(100,100,20,127,0,Brp4,127);}       // Slider sin pulsar

if(POSX>140 && POSX<210 && POSY>150 && POSY<220)
{       // Pulso en el encoder 4a (Pedal Sustain)
        X4a=POSX-175;Y4a=POSY-185;
        if(abs(X4a)>=12||abs(Y4a)>=12)
        {
                Th4a=Theta(X4a,Y4a);            // Calculo angulo
                Enc4a=Encoder(Th4a);            // Obtengo valor midi
        }
        ComDial(175,185,35,256,Th4a);
        comando=0xB3; // CC-Channel 4
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0x40; // CC64
        UCA0TXBUF=comando;
        __delay_cycles(10000);
```

```
        comando=Enc4a; // Valor Pedal (0bbbbbbb)
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}             // Dibujo el encoder pulsado (sin efecto 3D)
else
{ComDial(175,185,35,0,Th4a);}    // Encoder sin pulsar

if(POSX>230 && POSX<300 && POSY>150 && POSY<220)
{       // Pulso en el encoder 4b (Pan)
        X4b=POSX-265;Y4b=POSY-185;
        if(abs(X4b)>=12||abs(Y4b)>=12)
        {
                Th4b=Theta(X4b,Y4b);              // Calculo angulo
                Enc4b=Encoder(Th4b);              // Obtengo valor midi
        }
        ComDial(265,185,35,256,Th4b);
        comando=0xB3;  // CC-Channel 4
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=0xA;  // CC10
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Enc4b;// Valor Pan (0bbbbbbb)
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}             // Dibujo el encoder pulsado
else
{ComDial(265,185,35,0,Th4b);}    // Encoder sin pulsar

if(POSX>160 && POSX<190 && POSY>80 && POSY<100) // Programa<< (-1)
{
        ComButton(160,80,30,20,28,256,"<");     // Pinto boton pulsado
        Program4--; if (Program4<=0) Program4=0; // Program4 min = 0
        comando=0xC3; // Program Change-Channel 4
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Program4; // Data1
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}
else
{ ComButton(160,80,30,20,28,0,"<");}

if(POSX>250 && POSX<280 && POSY>80 && POSY<100) // Programa>> (+1)
{
        ComButton(250,80,30,20,28,256,">");
        Program4++; if (Program4>=127) Program4=127;// Program4 max = 127
        comando=0xC3; // Program Change-Channel 4
        UCA0TXBUF=comando;
        __delay_cycles(10000);
        comando=Program4; // Data1
        UCA0TXBUF=comando;
        __delay_cycles(1000000);
}
else
{ComButton(250,80,30,20,28,0,">");}
```

```
            ComColor(0x10,0x10,0x10);
            ComTXT(70,10, 28, OPT_CENTERX,"4");
            ComColor(0xff,0xff,0xff);
        }

        Dibuja();  // Ejecutar lista de comandos de la pantalla principal
    } // Fin else
  } // Fin while
} // Fin main
```

A continuación, se prosigue con las librerías del programa:

```
/*
 * HC05.h
 *
 */

#ifndef HC05_H_
#define HC05_H_

void InicializaHC05(void);

#endif /* HC05_H_ */


/*
 * HC05.c
 *
 */

#include "HC05.h"
#include <msp430.h>

void InicializaHC05(void)
{
          P1OUT = BIT3;  // Pin 3 es el boton del HC05
          P1REN = BIT3;  // Entrada con Pull up

          //UART HC05

          UCA0CTL1 |= UCSWRST;              // Reset UART
          UCA0CTL1 = UCSSEL_2 | UCSWRST;    // Reloj SMCLK

          UCA0MCTL = 0xAA;                  // Baudrate 115 200 bps
          UCA0BR0 = 0x45;
          UCA0BR1 = 0x0;

          UCA0CTL1 &= ~UCSWRST;             // Desactivar Reset

          do
             {
                 IFG1 &= ~OFIFG;      // Clear OSC fault flag
                 __delay_cycles(800); // 50us delay
             } while (IFG1 & OFIFG);
          IFG2 &= ~(UCA0RXIFG);
          IE2 |= UCA0RXIE;
}
```

```
/*
 * FT800.h
 *
 */
#define CMDBUF_SIZE           4096
#define CMD_APPEND            4294967070
#define CMD_BGCOLOR           4294967049
#define CMD_BITMAP_TRANSFORM  4294967073
#define CMD_BUTTON            4294967053
#define CMD_CALIBRATE         4294967061
#define CMD_CLOCK             4294967060
#define CMD_COLDSTART         4294967090
#define CMD_CRC               4294967043
#define CMD_DIAL              4294967085
#define CMD_DLSTART           4294967040
#define CMD_EXECUTE           4294967047
#define CMD_FGCOLOR           4294967050
#define CMD_GAUGE             4294967059
#define CMD_GETMATRIX         4294967091
#define CMD_GETPOINT          4294967048
#define CMD_GETPROPS          4294967077
#define CMD_GETPTR            4294967075
#define CMD_GRADCOLOR         4294967092
#define CMD_GRADIENT          4294967051
#define CMD_HAMMERAUX         4294967044
#define CMD_IDCT              4294967046
#define CMD_INFLATE           4294967074
#define CMD_INTERRUPT         4294967042
#define CMD_KEYS              4294967054
#define CMD_LOADIDENTITY      4294967078
#define CMD_LOADIMAGE         4294967076
#define CMD_LOGO              4294967089
#define CMD_MARCH             4294967045
#define CMD_MEMCPY            4294967069
#define CMD_MEMCRC            4294967064
#define CMD_MEMSET            4294967067
#define CMD_MEMWRITE          4294967066
#define CMD_MEMZERO           4294967068
#define CMD_NUMBER            4294967086
#define CMD_PROGRESS          4294967055
#define CMD_REGREAD           4294967065
#define CMD_ROTATE            4294967081
#define CMD_SCALE             4294967080
#define CMD_SCREENSAVER       4294967087
#define CMD_SCROLLBAR         4294967057
#define CMD_SETFONT           4294967083
#define CMD_SETMATRIX         4294967082
#define CMD_SKETCH            4294967088
#define CMD_SLIDER            4294967056
#define CMD_SNAPSHOT          4294967071
#define CMD_SPINNER           4294967062
#define CMD_STOP              4294967063
#define CMD_SWAP              4294967041
#define CMD_TEXT              4294967052
#define CMD_TOGGLE            4294967058
#define CMD_TOUCH_TRANSFORM   4294967072
#define CMD_TRACK             4294967084
#define CMD_TRANSLATE         4294967079
#define CMD_POINTSIZE          218103808
```

```
#define OPT_CENTER           1536
#define OPT_CENTERX          512
#define OPT_CENTERY          1024
#define OPT_FLAT             256
#define OPT_MONO             1
#define OPT_NOBACK           4096
#define OPT_NODL             2
#define OPT_NOHANDS          49152
#define OPT_NOHM             16384
#define OPT_NOPOINTER        16384
#define OPT_NOSECS           32768
#define OPT_NOTICKS          8192
#define OPT_RIGHTX           2048
#define OPT_SIGNED           256

#define CMD_BEGIN_BMP           0x1f000001
#define CMD_BEGIN_POINTS  0x1f000002
#define CMD_BEGIN_LINES   0x1f000003
#define CMD_BEGIN_LINESTRIP 0x1f000004
#define CMD_BEGIN_EDGESTRIP_R 0x1f000005
#define CMD_BEGIN_EDGESTRIP_L 0x1f000006
#define CMD_BEGIN_EDGESTRIP_A 0x1f000007
#define CMD_BEGIN_EDGESTRIP_B 0x1f000008
#define CMD_BEGIN_RECTS   0x1f000009
#define CMD_END                         0x21000000
#define CMD_DISPLAY 0
#define CMD_LINEWIDTH           0x0E000000   //ERROR EN LA DOCUMENTACION: PONE
0x06000000

#define RAM_CMD              1081344
#define RAM_DL               1048576
#define RAM_G               0
#define RAM_PAL              1056768
#define RAM_REG              1057792


#define REG_ANALOG           1058104
#define REG_ANA_COMP         1058160
#define REG_BIST_CMD         1058124
#define REG_BIST_EN          1058132
#define REG_BIST_RESULT      1058128
#define REG_BUSYBITS         1058008
#define REG_CLOCK            1057800
#define REG_CMD_DL           1058028
#define REG_CMD_READ         1058020
#define REG_CMD_WRITE        1058024
#define REG_CPURESET         1057820
#define REG_CRC              1058152
#define REG_CSPREAD          1057892
#define REG_CYA0             1058000
#define REG_CYA1             1058004
#define REG_CYA_TOUCH        1058100
#define REG_DATESTAMP        1058108
#define REG_DITHER           1057884
#define REG_DLSWAP           1057872
#define REG_FRAMES           1057796
#define REG_FREQUENCY        1057804
#define REG_GPIO             1057936
#define REG_GPIO_DIR         1057932
```

```
#define REG_HCYCLE            1057832
#define REG_HOFFSET           1057836
#define REG_HSIZE             1057840
#define REG_HSYNC0            1057844
#define REG_HSYNC1            1057848
#define REG_ID                1057792
#define REG_INT_EN            1057948
#define REG_INT_FLAGS         1057944
#define REG_INT_MASK          1057952
#define REG_MACRO_0           1057992
#define REG_MACRO_1           1057996
#define REG_MARCH_ACC         1058144
#define REG_MARCH_DIR         1058136
#define REG_MARCH_OP          1058140
#define REG_MARCH_WIDTH       1058148
#define REG_OUTBITS           1057880
#define REG_PCLK              1057900
#define REG_PCLK_POL          1057896
#define REG_PLAY              1057928
#define REG_PLAYBACK_FORMAT   1057972
#define REG_PLAYBACK_FREQ     1057968
#define REG_PLAYBACK_LENGTH   1057960
#define REG_PLAYBACK_LOOP     1057976
#define REG_PLAYBACK_PLAY     1057980
#define REG_PLAYBACK_READPTR  1057964
#define REG_PLAYBACK_START    1057956
#define REG_PWM_DUTY          1057988
#define REG_PWM_HZ            1057984
#define REG_RENDERMODE        1057808
#define REG_ROMSUB_SEL        1058016
#define REG_ROTATE            1057876
#define REG_SNAPSHOT          1057816
#define REG_SNAPY             1057812
#define REG_SOUND             1057924
#define REG_SWIZZLE           1057888
#define REG_TAG               1057912
#define REG_TAG_X             1057904
#define REG_TAG_Y             1057908
#define REG_TAP_CRC           1057824
#define REG_TAP_MASK          1057828
#define REG_TOUCH_ADC_MODE    1058036
#define REG_TOUCH_CHARGE      1058040
#define REG_TOUCH_DIRECT_XY   1058164
#define REG_TOUCH_DIRECT_Z1Z2  1058168
#define REG_TOUCH_MODE        1058032
#define REG_TOUCH_OVERSAMPLE  1058048
#define REG_TOUCH_RAW_XY      1058056
#define REG_TOUCH_RZ          1058060
#define REG_TOUCH_RZTHRESH    1058052
#define REG_TOUCH_SCREEN_XY   1058064
#define REG_TOUCH_SETTLE      1058044
#define REG_TOUCH_TAG         1058072
#define REG_TOUCH_TAG_XY      1058068
#define REG_TOUCH_TRANSFORM_A  1058076
#define REG_TOUCH_TRANSFORM_B  1058080
#define REG_TOUCH_TRANSFORM_C  1058084
#define REG_TOUCH_TRANSFORM_D  1058088
#define REG_TOUCH_TRANSFORM_E  1058092
#define REG_TOUCH_TRANSFORM_F  1058096
```

```
#define REG_TRACKER           1085440
#define REG_TRIM              1058156
#define REG_VCYCLE            1057852
#define REG_VOFFSET           1057856
#define REG_VOL_PB            1057916
#define REG_VOL_SOUND         1057920
#define REG_VSIZE             1057860
#define REG_VSYNC0            1057864
#define REG_VSYNC1            1057868

#define GRIS_CLARO 0xE0E0E0
#define GRIS_OSCURO 0x606060

#define S_TRIANG          0x04
#define S_BEEP            0x05
#define S_ALARM           0x06
#define S_WARBLE          0x07
#define S_PIPS            0x10
#define S_XILO            0x41
#define S_PIANO           0x46

#define N_DO              60
#define N_REB             61
#define N_RE              62
#define N_MIB             63
#define N_MI              64
#define N_FA              65
#define N_SOLB            66
#define N_SOL             67
#define N_LAB             68
#define N_LA              69
#define N_SIB             70
#define N_SI              71

#define      FT_GPU_INTERNAL_OSC 0x48 //default
#define      FT_GPU_EXTERNAL_OSC 0x44
#define      FT_GPU_PLL_48M 0x62  //default
#define      FT_GPU_PLL_36M 0x61
#define      FT_GPU_PLL_24M 0x64
#define      FT_GPU_ACTIVE_M   0x00
#define      FT_GPU_STANDBY_M  0x41//default
#define      FT_GPU_SLEEP_M    0x42
#define      FT_GPU_POWERDOWN_M  0x50
#define      FT_GPU_CORE_RESET   0x68

#define dword long
#define byte char


// Functions to specify the address before a read or write
void FT800_SPI_SendAddressWR(dword);
void FT800_SPI_SendAddressRD(dword);

// Functions to Read or Write data
char FT800_SPI_Read8(void);
long FT800_SPI_Read32(void);
void FT800_SPI_Write32(dword);
void FT800_SPI_Write16(unsigned int);
void FT800_SPI_Write8(byte);
```

```
// Functions specifically for host commands to the FT800
void FT800_SPI_HostCommand(byte);
void FT800_SPI_HostCommandDummyRead(void);

// Functions for creating Command Lists for the Co-Processor
unsigned int FT800_IncCMDOffset(unsigned int, byte);

// ******* Hardware Specific Functions *******
// MCU-specific functions

void HAL_Configure_MCU(void);
unsigned char HAL_SPI_ReadWrite(unsigned char);
void HAL_SPI_CSLow(void);
void HAL_SPI_CSHigh(void);
void HAL_SPI_PDlow(void);
void HAL_SPI_PDhigh(void);

void EscribeRam32(unsigned long dato);
void EscribeRam16(unsigned int dato);
void EscribeRam8(char dato);
void EscribeRamTxt(char* dato);
void Ejecuta_Lista(void);
void Dibuja(void);

unsigned long Lee_Reg(unsigned long dir);
void Esc_Reg(unsigned long dir, unsigned long valor);

void Inicia_pantalla(void);

void Comando(unsigned long COMM);
void ComEsperaFin(void);
void ComTXT(int x, int y, int fuente, int ops, char *cadena);
void ComNum(int x, int y, int fuente, int ops, unsigned long Num);
void ComTeclas(int x, int y, int w, int h, int fuente, unsigned int ops, char *Keys);
void ComVertex2ff(int x,int y);
void ComColor(int R, int G, int B);

void PadFIFO(void);
void espera(int T);
void Delay(void);
void Nueva_pantalla(unsigned long int color);

void Lee_pantalla(void);
void ComScrollbar(int x, int y, int w, int h, int ops, int val, int size, int range);
void ComFgcolor(int R, int G, int B);
void ComButton(int x, int y, int w, int h, int font, int ops, char *cadena);
void Espera_pant(void);
void Calibra_touch(void);

void ComGradient(int x0,int y0, long color0, int x1, int y1, long color1);
void ComDial(int x, int y, int r, int options, int val);
void ComSlider(int x, int y, int w, int h, int options, int val, int range);

unsigned int Theta(int x, int y);
unsigned int Encoder(int Th);
```

```
/*
 * HAL_msp430G2_FT800.c
 *
 */

#include "FT800.h"
#include <msp430.h>

unsigned int t;
// extern int Fin_Rx;
char Buffer_Rx;

// Hardware-Specific Functions. Explicacion funciones en capítulo 6.

void HAL_Configure_MCU(void)
{

            WDTCTL = WDTPW | WDTHOLD;

            P1SEL2 = BIT5 | BIT6 | BIT7 | BIT1 | BIT2;
            P1SEL = BIT5 | BIT6 | BIT7 | BIT1 | BIT2;
            P1DIR = BIT0 | BIT6 | BIT7;
            P1IES = 0; P1IFG = 0;

            P2OUT = 0;
            P2SEL = BIT4 | BIT6 | BIT7;
            P2DIR = BIT0 | BIT2 | BIT4 | BIT1;
            P2IES = 0; P2IFG = 0;

            BCSCTL2 = SELM_0 | DIVM_0 | DIVS_0;

          if (CALBC1_8MHZ != 0xFF)
          {
              /* Adjust this accordingly to your VCC rise time */
              __delay_cycles(100000);
              DCOCTL = 0x00;
              BCSCTL1 = CALBC1_8MHZ;      /* Set DCO to 8MHz */
              DCOCTL = CALDCO_8MHZ;
          }

          BCSCTL1 |= XT2OFF | DIVA_0;
          BCSCTL3 = XT2S_0 | LFXT1S_2 | XCAP_1;
          /* USCI B en modo SPI para la comunicacion*/

          UCB0CTL1 |= UCSWRST;
          UCB0CTL0 = UCCKPH | UCMSB | UCMST | UCMODE_0 | UCSYNC; //Config SPI
          UCB0CTL1 = UCSSEL_2 | UCSWRST; //Reloj SMCLK
          UCB0BR0 = 16; // 500kbps
          UCB0CTL1 &= ~UCSWRST;
          IFG2 &= ~(UCB0RXIFG);   // Borrar flag
          __bis_SR_register(GIE); // Habilitar GIE

       /*Timer 0: 1ms*/
      TA0CCTL0 = CM_0 | CCIS_0 | OUTMOD_0 | CCIE;
      TA0CCR0 = 11;
      TA0CTL = TASSEL_1 | ID_0 | MC_1;
}
```

```
unsigned char HAL_SPI_ReadWrite(unsigned char data)
{
       //Comunicacion bidireccional
       UCB0TXBUF = data;

       while (!(IFG2 & UCB0RXIFG));
       Buffer_Rx=UCB0RXBUF;

       return Buffer_Rx;
}

void HAL_SPI_CSLow(void)
{
    P2OUT&=~BIT2;
}

void HAL_SPI_CSHigh(void)
{
   P2OUT|=BIT2;
}

void HAL_SPI_PDlow(void)
{
    P2OUT&=~BIT1;
}

void HAL_SPI_PDhigh(void)
{
    P2OUT|=BIT1;
}

// General Functions

void Delay(void)
{
    unsigned int outer_count = 0x0000;
    unsigned int inner_count = 0x0000;

    outer_count = 0x0004;
    inner_count = 0xFFFF;

    while(outer_count > 0x0000)
    {
        while (inner_count > 0x0000)
        {
            inner_count --;
            asm(" NOP");
        }
    outer_count --;
    }
}

void espera(int T)
{
       t=0;
       while(t<T);
}
```

```
// Interrupcion Timer

#pragma vector=TIMER0_A0_VECTOR
__interrupt void TIMER0_A0_ISR_HOOK(void)
{
      t++;
}


/*
 * FT800.c
 *
 */


#include "FT800.h"
#include <msp430.h>

#define dword long
#define byte char


extern char chipid ;          // Holds value of Chip ID read from the FT800

unsigned long cmdBufferRd=0;  // Store the value read from the REG_CMD_READ register
unsigned long cmdBufferWr=0;  // Store the value read from the REG_CMD_WRITE register
unsigned int CMD_Offset;
unsigned long POSX, POSY, BufferXY;

//-----FT800 SPI Functions

// Send address of a register which is to be Written

void FT800_SPI_SendAddressWR(dword Memory_Address)
{
    byte SPI_Writebyte = 0x00;

    // Write out the address. Only the lower 3 bytes are sent, with the most
significant byte sent first
    SPI_Writebyte = ((Memory_Address & 0x00FF0000) >> 16);  // Mask off the first
byte to send
    SPI_Writebyte = (SPI_Writebyte & 0xBF | 0x80);          // Since this is a write,
the MSBs are forced to 10
    HAL_SPI_ReadWrite(SPI_Writebyte);                       // Call the low-level SPI
routine for this MCU to send this byte
//

    SPI_Writebyte = ((Memory_Address & 0x0000FF00) >> 8);   // Mask off the next byte
to be sent
    HAL_SPI_ReadWrite(SPI_Writebyte);                       // Call the low-level SPI
routine for this MCU to send this byte

    SPI_Writebyte = (Memory_Address & 0x000000FF);          // Mask off the next byte
to be sent (least significant byte)
    HAL_SPI_ReadWrite(SPI_Writebyte);                       // Call the low-level SPI
routine for this MCU to send this byte

}
```

```
// Send address of a register which is to be Read

void FT800_SPI_SendAddressRD(dword Memory_Address)
 {
      long porsi;
      porsi=Memory_Address;
    byte SPI_Writebyte = 0x00;

    // Write out the address. Only the lower 3 bytes are sent, with the most
significant byte sent first
    SPI_Writebyte = (char)((porsi & 0x00FF0000) >> 16);  // Mask off the first byte
to send
    SPI_Writebyte = (SPI_Writebyte & 0x3F);                // Since this is a read,
the upper two bits are forced to 00
    HAL_SPI_ReadWrite(SPI_Writebyte);                      // Call the low-level SPI
routine for this MCU to send this byte

    SPI_Writebyte = ((Memory_Address & 0x0000FF00) >> 8);   // Mask off the next byte
to be sent
    HAL_SPI_ReadWrite(SPI_Writebyte);                      // Call the low-level SPI
routine for this MCU to send this byte

    SPI_Writebyte = (Memory_Address & 0x000000FF);         // Mask off the next byte
to be sent (least significant byte)
    HAL_SPI_ReadWrite(SPI_Writebyte);                      // Call the low-level SPI
routine for this MCU to send this byte

    // Send dummy 00 as required in the FT800 datasheet when doing a read
    SPI_Writebyte = 0x00;                                  // Write dummy byte of 0
    HAL_SPI_ReadWrite(SPI_Writebyte);                      //
 }

// Write a 32-bit value

void FT800_SPI_Write32(dword SPIValue32)
{
    byte SPI_Writebyte = 0x00;

    SPI_Writebyte = (SPIValue32 & 0x000000FF);             //
    HAL_SPI_ReadWrite(SPI_Writebyte);                      // Write the first (least
significant) byte

    SPI_Writebyte = ((SPIValue32 & 0x0000FF00) >> 8);      //
    HAL_SPI_ReadWrite(SPI_Writebyte);                      //

    SPI_Writebyte = (SPIValue32 >> 16);                    //
    HAL_SPI_ReadWrite(SPI_Writebyte);                      //

    SPI_Writebyte = ((SPIValue32 & 0xFF000000) >> 24);     //
    HAL_SPI_ReadWrite(SPI_Writebyte);                      // Write the last (most
significant) byte
}
```

```
// Write a 16-bit value

void FT800_SPI_Write16(unsigned int SPIValue16)
{
    byte SPI_Writebyte = 0x00;

    SPI_Writebyte = (SPIValue16 & 0x00FF);                  //
    HAL_SPI_ReadWrite(SPI_Writebyte);                       // Write the first (least
significant) byte
    SPI_Writebyte = ((SPIValue16 & 0xFF00) >> 8);           //
    HAL_SPI_ReadWrite(SPI_Writebyte);                       // Write the last (most
significant) byte
}

// Write an 8-bit value

void FT800_SPI_Write8(byte SPIValue8)
{
    byte SPI_Writebyte = 0x00;
    // Write out the data
    SPI_Writebyte = (SPIValue8);                            //
    HAL_SPI_ReadWrite(SPI_Writebyte);                       // Write the data byte
}

// Read a 32-bit register

dword FT800_SPI_Read32()
 {
    byte SPI_Writebyte = 0x00;
    byte SPI_Readbyte = 0x00;
    dword SPI_DWordRead = 0x00000000;
    dword DwordTemp = 0x00000000;

    // Read the first data byte (this is the least significant byte of the register).
SPI writes out dummy byte of 0x00
    SPI_Writebyte = 0x00;                                   // Write a dummy 0x00
    SPI_Readbyte = HAL_SPI_ReadWrite(SPI_Writebyte);        // Get the byte which was
read by the low-level MCU routine
    DwordTemp = SPI_Readbyte;                               // Put received byte into
a temporary 32-bit value
    SPI_DWordRead = DwordTemp;                              // Put the byte into a
32-bit variable in the lower 8 bits

    // Read the actual data byte. We pass a 0x00 into the SPI routine since it always
writes when it reads
    SPI_Writebyte = 0x00;                                   // Write a dummy 0x00
    SPI_Readbyte = HAL_SPI_ReadWrite(SPI_Writebyte);        // Get the byte which was
read by the low-level MCU routine
    DwordTemp = SPI_Readbyte;                               // Put received byte into
a temporary 32-bit value
    SPI_DWordRead |= (DwordTemp << 8);                      // Put the byte into a
32-bit variable (shifted 8 bits up)

    // Read the actual data byte. We pass a 0x00 into the SPI routine since it always
writes when it reads
    SPI_Writebyte = 0x00;                                   // Write a dummy 0x00
    SPI_Readbyte = HAL_SPI_ReadWrite(SPI_Writebyte);        // Get the byte which was
read by the low-level MCU routine
```

```
        DwordTemp = SPI_Readbyte;                           // Put received byte into
a temporary 32-bit value
    SPI_DWordRead |= (DwordTemp << 16);                     // Put the byte into a
32-bit variable (shifted 16 bits up)

    // Read the actual data byte. We pass a 0x00 into the SPI routine since it always
writes when it reads
    SPI_Writebyte = 0x00;                                   // Write a dummy 0x00
    SPI_Readbyte = HAL_SPI_ReadWrite(SPI_Writebyte);        // Get the byte which was
read by the low-level MCU routine
        DwordTemp = SPI_Readbyte;                           // Put received byte into
a temporary 32-bit value
    SPI_DWordRead |= (DwordTemp << 24);                     // Put the byte into a
32-bit variable (shifted 24 bits up)

    // Return the byte which we read
    return(SPI_DWordRead);                                  //
}

// Read an 8-bit register

byte FT800_SPI_Read8()
 {
    byte SPI_Writebyte = 0x00;
    byte SPI_Readbyte = 0x00;

    // Read the data byte. We pass a 0x00 into the SPI routine since it always writes
when it reads
    SPI_Writebyte = 0x00;                                   // Write a dummy 0x00
    SPI_Readbyte = HAL_SPI_ReadWrite(SPI_Writebyte);        // Get the byte which was
read by the low-level MCU routine

    // Return the byte which we read
    return(SPI_Readbyte);                                   //
}

// Write a Host Command

void FT800_SPI_HostCommand(byte Host_Command)
{
    byte SPI_Writebyte = 0x00;

    // Chip Select Low
    HAL_SPI_CSLow();

    SPI_Writebyte = (Host_Command & 0x3F | 0x40);           // This is the command
being sent
    HAL_SPI_ReadWrite(SPI_Writebyte);                       //

    SPI_Writebyte = 0x00;                                   // Sending dummy byte
    HAL_SPI_ReadWrite(SPI_Writebyte);                       //


    SPI_Writebyte = 0x00;                                   // Sending dummy byte
    HAL_SPI_ReadWrite(SPI_Writebyte);                       //


    // Chip Select High
    HAL_SPI_CSHigh();
}
```

```
// Host Command Dummy Read

void FT800_SPI_HostCommandDummyRead(void)
 {
    byte SPI_Writebyte = 0x00;

    // Chip Select Low
    HAL_SPI_CSLow();
    // Read/Write
    SPI_Writebyte = 0x00;                                   // Sending dummy 00 byte
    HAL_SPI_ReadWrite(SPI_Writebyte);

    SPI_Writebyte = 0x00;                                   // Sending dummy 00 byte
    HAL_SPI_ReadWrite(SPI_Writebyte);

    SPI_Writebyte = 0x00;                                   // Sending dummy 00 byte
    HAL_SPI_ReadWrite(SPI_Writebyte);

    // Chip Select High
    HAL_SPI_CSHigh();
}

// Increment Command Offset

unsigned int FT800_IncCMDOffset(unsigned int Current_Offset, byte Command_Size)
{
    unsigned int New_Offset;

    New_Offset = Current_Offset + Command_Size;
    if(New_Offset > 4095)
    {
        New_Offset = (New_Offset - 4096);
    }
    return New_Offset;
}

// Obtencion posicion dial

unsigned int Theta(int x,int y)
{
      unsigned int Th;    // Angulo del dial
      int aux;            // tan(theta)

      if((x>=0)&&(y>=0)){ // Cuadrante IV
            aux=y/x;
            if(aux>=3.59){
                  Th=0x0000;}
            else if(aux>=1){
                  Th=0xEAA7;}
            else if(aux>=0.27){
                  Th=0xD552;}
            else{
                  Th=0xC000;}
      }
      if((x>=0)&&(y<=0)){// Cuadrante I
            aux=-(y/x);
            if(aux<=0.27){
                  Th=0xC000;}
            else if(aux<=1){
```

```
                Th=0xAAAA;}
            else if(aux<=3.59){
                Th=0x9553;}
            else{
                Th=0x8000;}
            }
        if((x<=0)&&(y<=0)){// Cuadrante II
            aux=y/x;
            if(aux>=3.59){
                Th=0x8000;}
            else if(aux>=1){
                Th=0x6AA9;}
            else if(aux>=0.27){
                Th=0x5555;}
            else{
                Th=0x4000;}
            }
        if((x<=0)&&(y>=0)){// Cuadrante III
            aux=-(y/x);
            if(aux<=0.27){
                Th=0x4000;}
            else if(aux<=1){
                Th=0x2AAA;}
            else if(aux>=0.27){
                Th=0x1555;}
            else{
                Th=0x0000;}
            }
        if((x==0)&&(y>=0)){Th=0x0000;} // Abajo
        if((x==0)&&(y<=0)){Th=0x8000;} // Arriba
        if((x>=0)&&(y==0)){Th=0xC000;} // Dcha
        if((x<=0)&&(y==0)){Th=0x4000;} // Izda

        return Th;
}

// Obtencion valor midi (0-127) de un encoder

unsigned int Encoder(int Th)
{
        unsigned int Enc;     // Valor MIDI para los encoder
        if(Th==0x0000)Enc=0;
        if(Th==0x1555)Enc=11;
        if(Th==0x2AAA)Enc=22;
        if(Th==0x4000)Enc=33;
        if(Th==0x5555)Enc=44;
        if(Th==0x6AA9)Enc=55;
        if(Th==0x8000)Enc=66;
        if(Th==0x9553)Enc=77;
        if(Th==0xAAAA)Enc=88;
        if(Th==0xC000)Enc=99;
        if(Th==0xD552)Enc=110;
        if(Th==0xEAA7)Enc=121;

        return Enc;
}
```

```
/*
 * ft800_commands.c
 *
 */


#include "FT800.h"
#include <msp430.h>

#define dword long
#define byte char

extern unsigned long POSX, POSY, BufferXY;
extern unsigned long POSYANT;
extern unsigned int CMD_Offset;

extern char chipid ;                    // Holds value of Chip ID read from the FT800

extern unsigned long cmdBufferRd;       // Store the value read from the REG_CMD_READ
register
extern unsigned long cmdBufferWr;       // Store the value read from the REG_CMD_WRITE
register

void ComEsperaFin(void)
{
do
{   //Esperar ejecucion de todos los comandos
    HAL_SPI_CSLow();                                //
    FT800_SPI_SendAddressRD(REG_CMD_WRITE);     //
    cmdBufferWr = FT800_SPI_Read32();           // Read the value of the
REG_CMD_WRITE register
    HAL_SPI_CSHigh();                               //
    HAL_SPI_CSLow();                                //
    FT800_SPI_SendAddressRD(REG_CMD_READ);      //
    cmdBufferRd = FT800_SPI_Read32();           // Read the value of the REG_CMD_READ
register
    HAL_SPI_CSHigh();                               //
} while(cmdBufferWr != cmdBufferRd);

// Check the actual value of the current WRITE register (pointer)
CMD_Offset = cmdBufferWr;
}


void EscribeRam32(unsigned long dato)
{
        HAL_SPI_CSLow();
        FT800_SPI_SendAddressWR(RAM_CMD + CMD_Offset);  // Writing to next available
location in FIFO (FIFO base address + offset)
        FT800_SPI_Write32(dato);                    // Vertex 2F    01XXXXXX XXXXXXXX
XYYYYYYY YYYYYYYY
        HAL_SPI_CSHigh();

        CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Move the CMD Offset since
we have just added 4 bytes to the FIFO
}
```

```
void EscribeRam16(unsigned int dato)
{
        HAL_SPI_CSLow();
        FT800_SPI_SendAddressWR(RAM_CMD + CMD_Offset);  // Writing to next available
location in FIFO (FIFO base address + offset)
        FT800_SPI_Write16(dato);                  // Vertex 2F    01XXXXXX XXXXXXXX
XYYYYYYY YYYYYYYY
        HAL_SPI_CSHigh();

        CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 2); // Move the CMD Offset since
we have just added 4 bytes to the FIFO
}

void EscribeRam8(char dato)
{
        HAL_SPI_CSLow();
        FT800_SPI_SendAddressWR(RAM_CMD + CMD_Offset);  // Writing to next available
location in FIFO (FIFO base address + offset)
        FT800_SPI_Write8(dato);                  // Vertex 2F    01XXXXXX XXXXXXXX
XYYYYYYY YYYYYYYY
        HAL_SPI_CSHigh();

        CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 1); // Move the CMD Offset since
we have just added 4 bytes to the FIFO
}

void EscribeRamTxt(char *cadena)
{
    while(*cadena) EscribeRam8(*cadena++);
    EscribeRam8(0);
}

void Comando(unsigned long COMM)
{
      HAL_SPI_CSLow();                                  //
      FT800_SPI_SendAddressWR(RAM_CMD + CMD_Offset);  // Writing to next available
location in FIFO (FIFO base address + offset)
      FT800_SPI_Write32(COMM);                  // Write the DL_START command
      HAL_SPI_CSHigh();                                 //

      CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Move the CMD Offset since
we have just added 4 bytes to the FIFO
}

void ComVertex2ff(int x,int y)
{
      long coord;
      coord=x;
      coord=coord<<19;
      coord=coord+ (y<<4);
       HAL_SPI_CSLow();                                 //
       FT800_SPI_SendAddressWR(RAM_CMD + CMD_Offset);  // Writing to next available
location in FIFO (FIFO base address + offset)
       FT800_SPI_Write32(0x40000000+coord);                  // Vertex 2F    01XXXXXX
XXXXXXXX XYYYYYYY YYYYYYYY
       HAL_SPI_CSHigh();                                //
       CMD_Offset = FT800_IncCMDOffset(CMD_Offset, 4); // Move the CMD Offset since
we have just added 4 bytes to the FIFO
}
```

```
void ComColor(int R, int G, int B)
{
	long color;
	color=R;
	color=color<<8;
	color+=G;
	color=color<<8;
	color+= B;
	// Color RGB   00000100 BBBBBBBB GGGGGGGG RRRRRRRR  (B/G/R = Colour values)
	Comando(0x04000000+color);
}

void ComTXT(int x, int y, int fuente, int ops, char *cadena)
{
	EscribeRam32(CMD_TEXT);
	EscribeRam16(x);
	EscribeRam16(y);
	EscribeRam16(fuente);
	EscribeRam16(ops);
	while(*cadena) EscribeRam8(*cadena++);
	EscribeRam8(0);
	PadFIFO();
}

void ComNum(int x, int y, int fuente, int ops, unsigned long Num)
{
	EscribeRam32(CMD_NUMBER);
	EscribeRam16(x);
	EscribeRam16(y);
	EscribeRam16(fuente);
	EscribeRam16(ops);
	EscribeRam32(Num);
	PadFIFO();
}

void ComTeclas(int x, int y, int w, int h, int fuente, unsigned int ops, char *Keys)
{
	EscribeRam32(CMD_KEYS);
	EscribeRam16(x);
	EscribeRam16(y);
	EscribeRam16(w);
	EscribeRam16(h);
	EscribeRam16(fuente);
	EscribeRam16(ops);
	while(*Keys) EscribeRam8(*Keys++);
	EscribeRam8(0);
	PadFIFO();
}

void Ejecuta_Lista(void)
{
HAL_SPI_CSLow();                              //
FT800_SPI_SendAddressWR(REG_CMD_WRITE);       //
FT800_SPI_Write16(CMD_Offset);                //
HAL_SPI_CSHigh();                             //
}
```

```
void PadFIFO(void)
{
	int i, pad;
	pad=CMD_Offset&0x0003;
	if(pad>0){
		for(i=0;i<4-pad;i++) EscribeRam8(0);
	}
}


void Dibuja(void)
{
	Comando(CMD_DISPLAY);
	Comando(CMD_SWAP);
	Ejecuta_Lista();
}


void Inicia_pantalla(void){
HAL_SPI_PDhigh();                               // Put the Power Down # pin high to
wake FT800
Delay();                                        // Delay for power up of regulator

FT800_SPI_HostCommandDummyRead();               // Read location 0 to wake up FT800
FT800_SPI_HostCommand(FT_GPU_EXTERNAL_OSC);     // Change the PLL to external clock -
optional
FT800_SPI_HostCommand(FT_GPU_PLL_48M);          //  Ensure configured to 48 MHz

// Note: Could now increase SPI clock rate up to 30MHz SPI if preferred

Delay();                                        // Delay
FT800_SPI_HostCommand(FT_GPU_CORE_RESET);       // Reset the core
Delay();                                        // Delay

FT800_SPI_HostCommand(FT_GPU_ACTIVE_M);         // Read address 0 to ensure FT800 is
active

chipid = 0x00;                                  // Read the Chip ID to check comms
with the FT800 - should be 0x7C
	while(chipid != 0x7C)
{
    HAL_SPI_CSLow();                            // CS low
    FT800_SPI_SendAddressRD(REG_ID);            // Send the address
    chipid = FT800_SPI_Read8();                 // Read the actual value
    HAL_SPI_CSHigh();
    Delay();// CS high
}

// ========================================================================
// Write the display registers on the FT800 for your particular display
// ========================================================================

HAL_SPI_CSLow();                                // CS low                   Write
REG_HCYCLE to 548
FT800_SPI_SendAddressWR(REG_HCYCLE);            // Send the address
FT800_SPI_Write16(408);                         // Send the 16-bit value
HAL_SPI_CSHigh();                               // CS high
```

```
HAL_SPI_CSLow();                          // CS low     Write REG_HOFFSET to 43
FT800_SPI_SendAddressWR(REG_HOFFSET);     // Send the address
FT800_SPI_Write16(70);                    // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_HSYNC0 to 0
FT800_SPI_SendAddressWR(REG_HSYNC0);      // Send the address
FT800_SPI_Write16(0);                     // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_HSYNC1 to 41
FT800_SPI_SendAddressWR(REG_HSYNC1);      // Send the address
FT800_SPI_Write16(10);                    // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_VCYCLE to 292
FT800_SPI_SendAddressWR(REG_VCYCLE);      // Send the address
FT800_SPI_Write16(263);                   // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_VOFFSET to 12
FT800_SPI_SendAddressWR(REG_VOFFSET);     // Send the address
FT800_SPI_Write16(13);                    // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_VSYNC0 to 0
FT800_SPI_SendAddressWR(REG_VSYNC0);      // Send the address
FT800_SPI_Write16(0);                     // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_VSYNC1 to 10
FT800_SPI_SendAddressWR(REG_VSYNC1);      // Send the address
FT800_SPI_Write16(2);                    // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_SWIZZLE to 2
FT800_SPI_SendAddressWR(REG_SWIZZLE);     // Send the address
FT800_SPI_Write16(2);                     // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_PCLK_POL to 1
FT800_SPI_SendAddressWR(REG_PCLK_POL);    // Send the address
FT800_SPI_Write16(0);                     // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_HSIZE to 480
FT800_SPI_SendAddressWR(REG_HSIZE);       // Send the address
FT800_SPI_Write16(320);                   // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_VSIZE to 272
FT800_SPI_SendAddressWR(REG_VSIZE);       // Send the address
FT800_SPI_Write16(240);                   // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high

HAL_SPI_CSLow();                          // CS low     Write REG_PCLK to 5
FT800_SPI_SendAddressWR(REG_PCLK);        // Send the address
FT800_SPI_Write16(5);                     // Send the 16-bit value
HAL_SPI_CSHigh();                         // CS high
```

```
// =========================================================================
// Configure the touch screen
// =========================================================================

HAL_SPI_CSLow();                                // CS low      Write the touch
threshold
FT800_SPI_SendAddressWR(REG_TOUCH_RZTHRESH);    // Send the address
FT800_SPI_Write16(1200);                        // Send the 16-bit value
HAL_SPI_CSHigh();                               // CS high

// =========================================================================
// Send an initial display list which will clear the screen to a black colour
// =========================================================================

// Set the colour which is used when the colour buffer is cleared
HAL_SPI_CSLow();                                //
FT800_SPI_SendAddressWR(RAM_DL + 0);            // Write first entry in Display List
(first location in DL Ram)
FT800_SPI_Write32(0x02000000);                  // Clear Color RGB   00000010
BBBBBBBB GGGGGGGG RRRRRRRR  (B/G/R = Colour values)
HAL_SPI_CSHigh();                               //

// Clear the Colour, Stencil and Tag buffers. This will set the screen to the 'clear'
colour set above.
HAL_SPI_CSLow();                                //
FT800_SPI_SendAddressWR(RAM_DL + 4);            // Write next entry in display list
(each command is 4 bytes)
FT800_SPI_Write32(0x26000007);                  // Clear 00100110 -------- -------- -
----CST  (C/S/T define which parameters to clear)
HAL_SPI_CSHigh();                               //

// Display command ends the display list
HAL_SPI_CSLow();                                //
FT800_SPI_SendAddressWR(RAM_DL + 8);            // Write next entry in display list
FT800_SPI_Write32(0x00000000);                  // DISPLAY command 00000000 00000000
00000000 00000000
HAL_SPI_CSHigh();                               //

// Writing to the DL_SWAP register tells the Display Engine to render the new screen
designed above.
HAL_SPI_CSLow();                                //
FT800_SPI_SendAddressWR(REG_DLSWAP);            // Writing to the DL_SWAP
register...value 10 means render after last frame complete
FT800_SPI_Write32(0x00000002);                  // 00000000 00000000 00000000
000000SS  (SS bits define when render occurs)
HAL_SPI_CSHigh();                               // CS high


// =========================================================================
// Set the GPIO pins of the FT800 to enable the display now
// =========================================================================

HAL_SPI_CSLow();                                // CS low        Write REG_GPIO_DIR
FT800_SPI_SendAddressWR(REG_GPIO_DIR);          // Send the address
FT800_SPI_Write8(0x83);                         // Send the 8-bit value
HAL_SPI_CSHigh();                               // CS high
```

```
HAL_SPI_CSLow();                                    // CS low      Write REG_GPIO
FT800_SPI_SendAddressWR(REG_GPIO);                  // Send the address
FT800_SPI_Write8(0x83);                             // Send the 8-bit value
HAL_SPI_CSHigh();                                   // CS high
}

void Nueva_pantalla(unsigned long int color)
{
    ComEsperaFin();
    Comando(CMD_DLSTART);
      Comando(0x02000000+color);  // Clear Color RGB   00000010 BBBBBBBB GGGGGGGG
RRRRRRRR  (B/G/R = Colour values)
      Comando(0x26000007);                          // Clear 00100110 -------- ---
----- -----CST  (C/S/T define which parameters to clear))
}


void Lee_pantalla(void)
{
            HAL_SPI_CSLow();
            FT800_SPI_SendAddressRD(REG_TOUCH_SCREEN_XY);
            BufferXY = FT800_SPI_Read32();          // Lee valor pantalla tactil
            HAL_SPI_CSHigh();
            POSX=(BufferXY&0xffff0000)>>16;
            POSY=BufferXY&0x0000FFFF;
}

unsigned long Lee_Reg(unsigned long dir)
{
      unsigned long Buff_temp;
      HAL_SPI_CSLow();
      FT800_SPI_SendAddressRD(dir);
      Buff_temp = FT800_SPI_Read32();
      HAL_SPI_CSHigh();
      return(Buff_temp);
}

void Esc_Reg(unsigned long dir, unsigned long valor)
{
        HAL_SPI_CSLow();
        FT800_SPI_SendAddressWR(dir);
        FT800_SPI_Write32(valor);
        HAL_SPI_CSHigh();
}


void ComScrollbar(int x, int y, int w, int h, int ops, int val, int size, int range)
{
      EscribeRam32(CMD_SCROLLBAR);
      EscribeRam16(x);
      EscribeRam16(y);
      EscribeRam16(w);
      EscribeRam16(h);
      EscribeRam16(ops);
      EscribeRam16(val);
      EscribeRam16(size);
      EscribeRam16(range);
}
```

```
void ComFgcolor(int R, int G, int B)
{
	long color;
	color=R;
	color=color<<8;
	color+=G;
	color=color<<8;
	color+= B;

	EscribeRam32 (CMD_FGCOLOR);
	EscribeRam32 (color);
}

void ComButton(int x, int y, int w, int h, int font, int ops, char *cadena)
{
	EscribeRam32(CMD_BUTTON);
	EscribeRam16(x);
	EscribeRam16(y);
	EscribeRam16(w);
	EscribeRam16(h);
	EscribeRam16(font);
	EscribeRam16(ops);
	while(*cadena) EscribeRam8(*cadena++);
	EscribeRam8(0);
	PadFIFO();
}

void ComDial(int x, int y, int r, int options, int val)
{
	EscribeRam32(CMD_DIAL);
	EscribeRam16(x);
	EscribeRam16(y);
	EscribeRam16(r);
	EscribeRam16(options);
	EscribeRam16(val);
	EscribeRam16(0);
}

void ComSlider(int x, int y, int w, int h, int options, int val, int range)
{
		EscribeRam32(CMD_SLIDER);
	    EscribeRam16(x);
		EscribeRam16(y);
	    EscribeRam16(w);
		EscribeRam16(h);
		EscribeRam16(options);
		EscribeRam16(val);
		EscribeRam16(range);
		EscribeRam16(0);
}
```

```
void Calibra_touch(void)
{
        ComEsperaFin();
        Comando(CMD_DLSTART);
        Comando(0x02f7f7f7);  // Clear Color RGB   00000010 BBBBBBBB GGGGGGGG
RRRRRRRR  (B/G/R = Colour values)
        Comando(0x26000007);                              // Clear 00100110 -------- --
------ -----CST  (C/S/T define which parameters to clear))
        ComColor(0x00,0x00,0x00);

         ComTXT(60,30,27,0,"Pulsa en el punto");
         EscribeRam32(CMD_CALIBRATE);

         Dibuja();

         espera(500);
         ComEsperaFin();
         Comando(CMD_DLSTART);
         Comando(0x02ffff00);  // Clear Color RGB   00000010 BBBBBBBB GGGGGGGG
RRRRRRRR  (B/G/R = Colour values)
         Comando(0x26000007);                              // Clear 00100110 -------- --
------ -----CST  (C/S/T define which parameters to clear))
         Dibuja();
}

void ComGradient(int x0,int y0, long color0, int x1, int y1, long color1)
{
       EscribeRam32(CMD_GRADIENT);
       EscribeRam16(x0);
       EscribeRam16(y0);
       EscribeRam32(color0);
       EscribeRam16(x1);
       EscribeRam16(y1);
       EscribeRam32(color1);
}

void Espera_pant(void)
{
       do{
             Lee_pantalla();
         }while(POSY==0x8000);
}
```