Diseño y desarrollo de un simulador de comunicaciones RS-232 y RS-422 basado en Microcontrolador.

Carlos I. Ayllón Fernández

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación.
Desarrollo de Aplicaciones Electrónicas.

Nombre Tutor/a de TF
Aleix López Antón

Profesor/a responsable de la asignatura
Carlos Monzo Sánchez

Fecha Entrega
Junio de 2023

Universitat Oberta de Catalunya

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Diseño y desarrollo de un simulador de comunicaciones RS-232 y RS-422 basado en Microcontrolador*
Nombre del autor:	*Carlos Ignacio Ayllón Fernández*
Nombre del consultor/a:	*Aleix López Antón*
Nombre del PRA:	*Carlos Monzo Sánchez*
Fecha de entrega (mm/aaaa):	*06/2023*
Titulación o programa:	Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Área del Trabajo Final:	*Desarrollo de aplicaciones electrónicas*
Idioma del trabajo:	*Castellano*
Palabras clave	*Simulador, RS-232, RS-422, Microcontrolador, Comunicaciones*

Resumen del Trabajo

En el ámbito de las comunicaciones serie se emplean desde hace décadas los protocolos RS-232 y RS-422. Sin embargo no abundan las herramientas que simulen y generen estas señales con precisión, algo que es extremadamente útil a la hora de desarrollar o comprobar dispositivos electrónicos. Las herramientas disponibles para estos fines, o tienen un elevado coste, o en su mayoría están basadas en software y conversores USB, lo que hace que sean poco deterministas, imprecisas o no alcancen las altas velocidades empleadas hoy en día.

Este trabajo fin de grado pretende suplir esta necesidad, generando un producto compuesto por una parte hardware y otra software. En él se describe la planificación, el diseño, desarrollo y pruebas funcionales de un simulador que genera las señales de comunicaciones serie de dichos estándares. Para ello se ha diseñado y desarrollado la electrónica, el circuito impreso y la programación de un firmware en un microcontrolador PIC32MX, que genera todas las señales descritas.

Junto al hardware, se ha desarrollado un software de aplicación para PC, que sirve para configurar los diferentes parámetros, como los mensajes a enviar, las velocidades, numero de bits, retardos etc. Tras la fabricación del prototipo y la verificación de las señales generadas, se evalúan los resultados y se analizan las posibles líneas de trabajo futuro a seguir.

Abstract

RS-232 and RS-422 protocols have been used in the field of serial communications for decades. However, there are not many tools that accurately simulate and generate these signals, which is extremely useful when developing or testing electronic devices. The tools available for these purposes are either expensive or mostly based on software and USB converters, which makes them not very deterministic, inaccurate or not reaching the high speeds used today.

This final degree work aims to meet this need, generating a product composed of a hardware and a software part. It describes the planning, design, development and functional testing of a simulator that generates serial communications signals of these standards. For this purpose, the electronics, the printed circuit board and the programming of a firmware in a PIC32MX microcontroller, which generates all the described signals, have been designed and developed.

Together with the hardware, a PC application software has been developed, which is used to configure the different parameters, such as the messages to be sent, speeds, number of bits, delays, etc. After manufacturing de prototype and verifying the signals, the results are evaluated and the possible lines of future work are analyzed.

Índice

Lista de figuras

Lista de Tablas

1. Introducción

La norma del primer estándar de comunicaciones serie RS-232 -designado oficialmente como *TIA/EIA-232* [1]- se publicó hace ya más de 60 años. A pesar de su antigüedad este tipo de comunicaciones no han dejado de emplearse y no han perdido aplicación ni utilidad. Junto con su predecesor diferencial RS-422 ha sido y es habitualmente incorporado en equipamiento electrónico de diseño reciente que requiere comunicaciones asíncronas, destacando por su simplicidad y fiabilidad, que ha hecho que sean ampliamente utilizados en el ámbito industrial y de las comunicaciones.

Sin embargo, las herramientas de desarrollo y soporte disponibles no han evolucionado a la par de los sistemas operativos o requerimientos modernos y se hace difícil encontrar herramientas de simulación asequibles y deterministas, que sean capaces de reproducir fielmente esta transmisión a los altos ratios de bits empleados hoy día en muchos sistemas.

1.1. Contexto y justificación del Trabajo

La mayoría de las herramientas de desarrollo y simulación de comunicaciones serie disponibles en el mercado son soluciones basadas mayoritariamente en *software* -y dado que la práctica totalidad de los ordenadores modernos ya no disponen de puerto serie físico- estos se emplean junto con conversores de puerto USB a RS-232 o RS-422 -que si bien son capaces de reproducir y enviar los mensajes- no suelen ser deterministas en cuanto a *jitter*, retardos, separación entre mensajes y otras casuísticas específicas, debido principalmente a que estas aplicaciones no realizan un control directo del hardware a bajo nivel, lo que hace que no sean útiles o precisos para el fiel desarrollo y prueba de muchos sistemas que requieren comunicaciones precisas.

Este proyecto pretende cubrir esta necesidad y desarrollar un simulador *hardware* de comunicaciones serie RS-232 y RS-422 específico para estos propósitos, basado en un microcontrolador, que sea capaz de reproducir fielmente las tramas habitualmente empleadas por estos protocolos de comunicaciones en las diferentes aplicaciones en las que se pueden encontrar hoy en día. Este desarrollo seria de utilidad -entre otros-, para comprobar equipos de adquisición de datos, desarrollar equipos de comunicaciones o auditar el funcionamiento de dispositivos electrónicos.

Además del diseño y desarrollo del hardware de interfaz y de la programación del microcontrolador para efectuar estas labores, en el proyecto se desarrollará también un software de control en forma de aplicación de escritorio para PC, que interactuará con el *firmware* del microcontrolador y con el que se definirá la configuración de este, para establecer los diferentes campos de utilidad, como el mensaje a enviar y su longitud, la velocidad, bits o paridad que va a reproducir el hardware anteriormente descrito.

1.2. Objetivos del Trabajo

Aplicando todos los conocimientos obtenidos durante los estudios de Grado, el objetivo del proyecto es desarrollar una solución completa *Hardware/Software* que ofrezca unos mínimos de precisión e invariabilidad a la hora de simular señales digitales asíncronas de los protocolos RS232 y RS422. Como meta se pretende poder enviar mensajes de una velocidad de hasta 1 Mbps, pudiendo enviar mensajes de longitud definida de hasta 1024 Bytes, con 8 o 9 bits de datos, con o sin paridad y con 1 o 2 bits de parada. Al tratarse de un protocolo asíncrono el error máximo admisible en el *baudrate* no debería superar el 1% para garantizar un mínimo error en la comunicación.

Se pretende que el simulador construido no solo sea capaz de enviar mensajes unitarios, sino que también pueda enviar mensajes en bucle, de manera indefinida y autónoma y que los propios mensajes puedan contener un contador incremental que identifique fácilmente cualquier pérdida de mensajes en las pruebas de validación.

Igualmente se deberá poder establecer un tiempo definido de separación o *gap* entre los mensajes, con una resolución y variabilidad adecuada al *baudrate* establecido en la configuración. Esta configuración -una vez establecida en el software- deberá ser posible almacenarla en una memoria no volátil, de forma que una vez configurado el dispositivo hardware, éste pueda funcionar de forma independiente con la última configuración establecida y sin necesidad de nueva conexión con el software, lo que facilitaría la portabilidad del dispositivo y su uso independiente, tanto en laboratorio como en entornos de campo.

A continuación se concretan los diferentes objetivos y requisitos de la parte Hardware y Software:

- Parte Hardware: Microcontrolador e interfaces RS-232/RS-422
  - Estudio de placas de evaluación de microcontrolador PIC32 adecuadas.
  - Análisis, cálculos, selección de medios, componentes y presupuesto necesario para llevar a cabo el proyecto.
  - Diseño de un circuito electrónico y su correspondiente circuito impreso, que haga de interfaz entre las señales del microcontrolador y los puertos de salida RS-232 y RS-422. Para ello se emplearán los circuitos integrados y componentes electrónicos seleccionados, el sistema también incluirá un *display LCD* donde se podrán mostrar configuraciones, condiciones de error o status.
  - Se fabricará un prototipo funcional de PCB que -conectado a la placa del microcontrolador- sirva de modelo para la evaluación práctica del proyecto.
  - En el apartado de programación del microcontrolador se desarrollará un *firmware* de control en lenguaje C que deberá, por un lado interactuar con la parte software del proyecto y recibir los parámetros a simular por parte del usuario/a, almacenándolos en una memoria no volátil, y por otro lado establecerá estas configuraciones para poder generar los mensajes y señales oportunas por los puertos RS-

232 o RS-422, según se requiera. Durante el desarrollo del firmware se alcanzarán los siguientes objetivos:

- Dominar el entorno de desarrollo MPLAB X de Microchip y su compilador para configurar pines y registros de control del PIC32.
- Conseguir recibir parámetros y mensajes desde PC al PIC por puerto USB.
- Almacenar estos parámetros y datos en memoria no volátil.
- Leer estos datos y la configuración de la memoria no volátil, incluso una vez desconectado del PC y reiniciado el HW.
- Mostrar mensajes de configuración y status en la pantalla LCD.
- Configurar las UART del PIC con los parámetros necesarios.
- Generar los mensajes en las UART con los datos deseados.

- Parte Software -Programa de control y configuración para PC-

  - Utilizando el lenguaje de programación e IDE *Processing*, se programará una aplicación de escritorio que correrá en un PC y con la que se definan los parámetros a simular. Contendrá las diferentes opciones seleccionables por el usuario/a, tales como:
    - Selección del puerto HW a emplear, entre RS-232 o RS-422.
    - Campo de texto del mensaje a enviar hasta 1024 caracteres.
    - Enviar el mensaje codificado en ASCII o en hexadecimal.
    - Mensaje en bucle repetitivo con *gap* de separación a elegir.
    - Posibilidad de incluir contador de mensajes en el propio mensaje.
    - *Baudrate* seleccionable entre 2400 bps y 1 Mbps.
    - 8 o 9 bits de datos
    - Paridad par/impar o sin paridad
    - 1 o 2 bits de stop

  - Este programa se deberá comunicar con el microcontrolador por un puerto USB y enviará a éste todos los parámetros configurados para iniciar la simulación en la parte del hardware. Se desarrollarán los

mensajes de comunicación que enviará la aplicación y que interpretará el PIC para incorporar la configuración.

Para llevar todo esto a buen fin se van a poner en práctica todos los conocimientos de ingeniería adquiridos en las asignaturas del Grado. Se van a aplicar competencias muy diversas, que van desde los conocimientos de diseño electrónico y de *PCB´s*, análisis y cálculos diversos, programación, lógica, depuración y resolución de problemas, montaje de circuitos y todo en conjunción con otras materias transversales como la gestión de proyectos, documentación, interpretación de *datasheets* o recursos en idioma extranjero, etc.

Una vez fabricado el prototipo se realizarán las diferentes pruebas funcionales necesarias para verificar las capacidades de simulación proyectadas y en caso de ser necesario, detectar posibles mejoras o comportamientos a depurar.

1.3. Impacto en sostenibilidad, ético-social y de diversidad

En este trabajo no se han identificado impactos, ni negativos ni positivos, en cuanto a sostenibilidad, pues el producto final no tiene una dimensión de producción como para afectar significativamente en consumos, reciclaje o impacto ambiental. De igual manera, tampoco se ha identificado impacto alguno en los aspectos ético-sociales, pues no se estima se pueda hacer ningún uso fraudulento con él, ni que pueda afectar a seguridad, derechos u otros. En el aspecto de diversidad, género o derechos humanos tampoco se han identificado impactos, siendo un trabajo lo suficientemente técnico como para que no exista en el ningún tipo de limitación referente a aspectos de género, raza, religión etc.

1.4. Enfoque y método seguido

En este trabajo se va a desarrollar un prototipo de producto nuevo, utilizando por un lado una de las placas de desarrollo para microcontrolador existente en

el mercado como producto comercial, y por otro lado un hardware y software diseñado a medida.

Esta estrategia permitirá acelerar el proceso de desarrollo utilizando un producto maduro -como es la placa de desarrollo- y personalizándolo con el diseño a medida, que conseguirá alcanzar los objetivos del proyecto minimizando el tiempo y los riesgos.

1.5. Motivación

Para el autor es un reto el emplear todos los conocimientos de ingeniería adquiridos durante el grado, en un entorno teórico-práctico y orientado a desarrollar un producto de utilidad final, en el que se tratan en profundidad las comunicaciones, tanto de los puertos RS232/422 como la propia comunicación entre una aplicación para PC y un hardware por un puerto USB, así como la programación de microcontroladores.

Además, se estima que este proyecto tiene una utilidad real más allá de un ensayo, pues aunque en el mercado es posible encontrar varios dispositivos, como *sniffers,* analizadores o convertidores *virtualizados* por software, es difícil encontrar soluciones hardware de bajo coste, que puedan reproducir fielmente las señales de estos puertos a altas velocidades y con control directo del hardware. Sería muy útil un dispositivo así para verificar desde sistemas de adquisición de datos que capturan estas señales, hasta equipos de ámbito industrial, de GPS etc. garantizando que las señales simuladas van a ser generadas con precisión, sin depender del sistema operativo usado, de las capacidades del PC, puertos disponibles, conversores etc. En este sentido, a pesar de que los puertos de este tipo parecerían estar obsoletos, se estima que es un proyecto innovador, teniendo en cuenta que no es fácil disponer de herramientas similares.

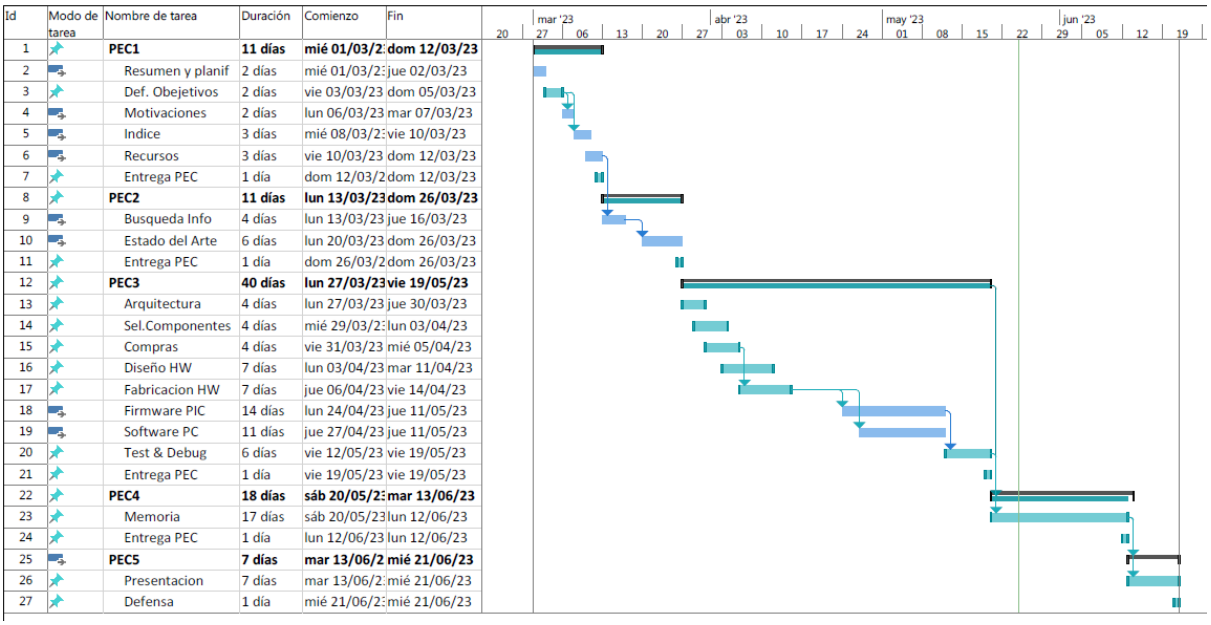
Igualmente, se aprecia especial complejidad en desarrollar en paralelo un software de control y el firmware asociado en el hardware, con las dificultades de depuración que puede conllevar tener dos sistemas en desarrollo

comunicándose, funcionando en paralelo y que tienen que interactuar, y esto sin duda es un reto para poner en valor recursos y conocimientos.

Otra motivación es desarrollar y ampliar los conocimientos en cuanto a electrónica, *PCB´s* y programación de microcontroladores PIC, junto al uso del entorno de desarrollo MPLAB y su compilador. Estos microcontroladores tienen una versatilidad y capacidad amplia y este proyecto puede ser la base de desarrollos futuros con dispositivos modernos, junto a otros desarrollos electrónicos, estableciendo bases en un entorno real y escalable. De igual manera el realizar una aplicación *desktop* con *Processing* para realizar control de hardware sin duda es una forma de dominar nuevos entornos de interacción.

1.6. Planificación del Trabajo

La planificación del trabajo se ha realizado siguiendo las entregas parciales de evaluación del proyecto y se ha realizado el siguiente diagrama de Gantt de la figura 1 contando aproximadamente con 4 meses para realizar todas las fases.



Id	Modo de tarea	Nombre de tarea	Duración	Comienzo	Fin
1		**PEC1**	**11 días**	**mié 01/03/2**	**dom 12/03/23**
2		Resumen y planif	2 días	mié 01/03/2	jue 02/03/23
3		Def. Obejetivos	2 días	vie 03/03/23	dom 05/03/23
4		Motivaciones	2 días	lun 06/03/23	mar 07/03/23
5		Indice	3 días	mié 08/03/2	vie 10/03/23
6		Recursos	3 días	vie 10/03/23	dom 12/03/23
7		Entrega PEC	1 día	dom 12/03/2	dom 12/03/23
8		**PEC2**	**11 días**	**lun 13/03/23**	**dom 26/03/23**
9		Busqueda Info	4 días	lun 13/03/23	jue 16/03/23
10		Estado del Arte	6 días	lun 20/03/23	dom 26/03/23
11		Entrega PEC	1 día	dom 26/03/2	dom 26/03/23
12		**PEC3**	**40 días**	**lun 27/03/23**	**vie 19/05/23**
13		Arquitectura	4 días	lun 27/03/23	jue 30/03/23
14		Sel.Componentes	4 días	mié 29/03/2	lun 03/04/23
15		Compras	4 días	vie 31/03/23	mié 05/04/23
16		Diseño HW	7 días	lun 03/04/23	mar 11/04/23
17		Fabricacion HW	7 días	jue 06/04/23	vie 14/04/23
18		Firmware PIC	14 días	lun 24/04/23	jue 11/05/23
19		Software PC	11 días	jue 27/04/23	jue 11/05/23
20		Test & Debug	6 días	vie 12/05/23	vie 19/05/23
21		Entrega PEC	1 día	vie 19/05/23	vie 19/05/23
22		**PEC4**	**18 días**	**sáb 20/05/2**	**mar 13/06/23**
23		Memoria	17 días	sáb 20/05/23	lun 12/06/23
24		Entrega PEC	1 día	lun 12/06/23	lun 12/06/23
25		**PEC5**	**7 días**	**mar 13/06/2**	**mié 21/06/23**
26		Presentacion	7 días	mar 13/06/2	mié 21/06/23
27		Defensa	1 día	mié 21/06/2	mié 21/06/23

Figura 1: Diagrama de Gantt de la planificación.

1.7. Presupuesto económico

Se ha establecido un presupuesto de gastos para el proyecto de aproximadamente 150 Euros, que se distribuirán principalmente en las compras de componentes electrónicos, placa de desarrollo, programador, PCB, display, conectores etc.

Los programas utilizados serán de libre distribución, o en el caso de programas comerciales se utilizarán con licencias de uso estudiantil o de prueba gratuita limitada según los términos de sus respectivas licencias.

El tiempo o mano de obra empleado en todo el proyecto se estima aproximadamente en 400 horas, asumidas enteramente por el autor y que no se van a cuantificar económicamente pues -al tratarse de un trabajo universitario sin fines de comercialización- no se espera un retorno a la inversión ni una amortización del desarrollo o de los útiles adquiridos, lo cual sin duda requeriría un análisis más pormenorizado de costes, beneficios, número de unidades a producir, plazos, márgenes y otros.

1.8. Análisis de viabilidad y riesgos

Una vez establecidas las bases del trabajo, los objetivos y requisitos de diseño junto con el tiempo disponible para su realización se ha evaluado si es factible su éxito, tanto a nivel técnico como con los medios y recursos disponibles.

Realizando un estudio preliminar de las placas de evaluación, así como las características técnicas de los componentes disponibles en el mercado se estima que la familia de microcontroladores PIC32MX dispone de las características para hacer viable el proyecto técnicamente, así como la disponibilidad de otros componentes junto con las herramientas y útiles necesarios.

No obstante el proyecto está limitado en tiempo y coste por lo que no está exento de riesgos, que se han valorado y establecido, así como los planes de contingencia que se detallan en las tablas 1 y 2:

INCIDENCIAS Y RIESGOS				
Código	*Descripción*	*Causa*	*Probabilidad*	*Impacto*
R01	Pérdida de documentos.	Errores al guardar o avería disco duro.	Baja	Alto
R02	Avería en PC	Deja de funcionar	Baja	Medio
R03	Avería en PIC	Placa de desarrollo deja de funcionar	Media	Alto
R04	Avería componentes	Componentes electrónicos fallan	Media	Alto
R05	Problemas de software	Mal funcionamiento en IDE´s u otros programas	Baja	Alto
R06	Fallos en PCB	Fallos de diseño o fabricación en PCB	Media	Alto

Tabla 1: Incidencias y riesgos

PLAN DE CONTINGENCIAS		
Código	*Tipo de acción*	*Acción*
A01R01	Mitigadora	Realizar copias de seguridad diarias en medios externos, guardar versión de trabajo regularmente, control de versiones.
A01R02	Mitigadora	Disponer de otro PC auxiliar
A01R03	Correctora	Adquisición urgente de otra placa
A02R03	Correctora	Reprogramar calendario y avanzar otros trabajos
A01R04	Mitigadora	Al comprar los componentes adquirir repuestos
A02R04	Correctora	Sustituir componentes y reprogramar trabajos
A01R05	Correctora	Instalar el software en PC de respaldo
A01R06	Mitigadora	Doble check previo de los diseños y trabajos
A02R06	Correctora	Cablear o reparar fallos

Tabla 2: Plan de contingencias

1.9. Breve sumario de productos obtenidos

En el diagrama WBS de la figura 2 se muestra un resumen de los principales entregables del proyecto, que incluirá el prototipo, el firmware y los esquemáticos de la parte hardware, la aplicación de control por PC en la parte software y la documentación correspondiente al Trabajo Fin de Grado.

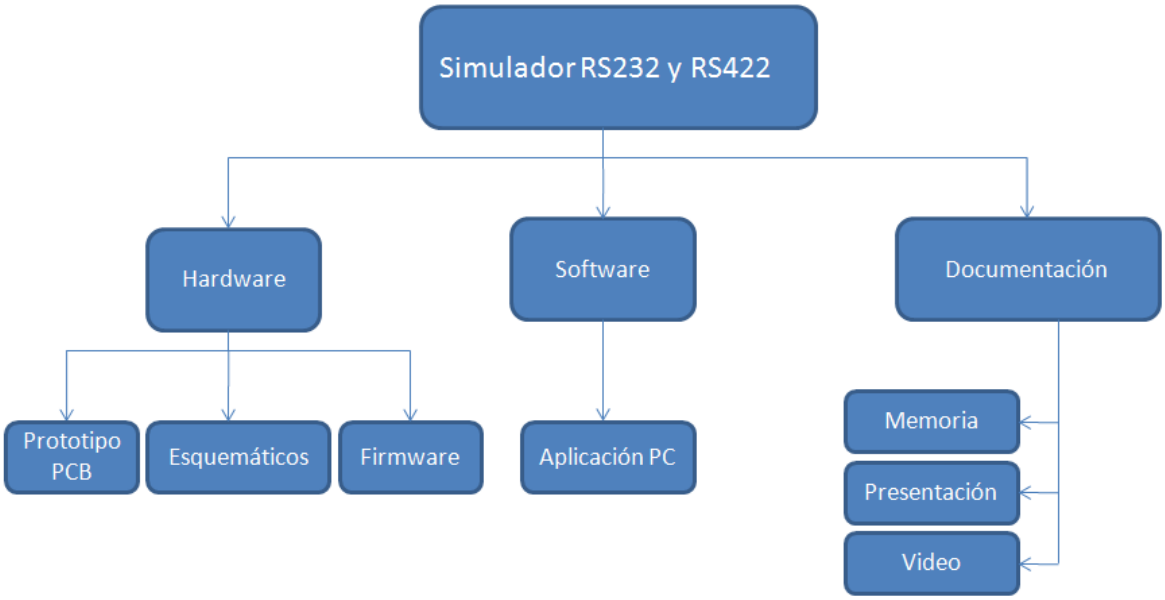


Figura 2: Diagrama WBS**.**

1.10. Breve descripción de los otros capítulos de la memoria

Además de la introducción del capítulo 1, esta memoria se encuentra estructurada en los siguientes capítulos:

- Capítulo 2: Estado del Arte. En este capítulo se detallan los estándares y sistemas disponibles en el mercado.
- Capítulo 3: Arquitectura del Sistema. En este capítulo se detallan todos los puntos relativos al diseño y fabricación del sistema, tanto del hardware como del software.
- Capítulo 4: Verificación del sistema a cadena completa Hw/Sw. En este capítulo se realizan todas las pruebas de verificación e integración del conjunto hardware/software y el análisis de resultados.
- Capítulo 5: Conclusiones: Se evalúa la consecución los objetivos y se analizan líneas futuras.
- Capítulo 6: Glosario de términos.
- Capítulo 7: Bibliografía utilizada.
- Capítulo 8: Anexos. Contiene esquemáticos del diseño, código fuente y otros documentos reseñables.

2. Estado del Arte

2.1. Estándares EIA/TIA

Los estándares aplicables a las comunicaciones serie tratadas en este trabajo son los denominados *EIA/TIA-232* [1] y el *EIA/TIA-422* [2]. El primero de ellos -habitualmente conocido como *recomended standard 232* ó RS-232- se desarrolló por la Asociación de Industrias Electrónicas (*EIA*) a principios de los años 60 del siglo pasado, en un esfuerzo por estandarizar la interfaz entre equipos de comunicaciones. Aunque entonces se enfocó a la conexión entre un módem y un terminal, enseguida otras aplicaciones adoptaron el estándar 232.

El creciente uso de los computadores personales (PC) rápidamente aseguró que RS-232 se convirtiera en el estándar de la industria para todas las interfaces seriales de bajo costo. El estándar ha sufrido varias revisiones a lo largo de los años, la última revisión disponible –la revisión F- fue aprobada en 1997 y aunque no se realizaron grandes modificaciones se publicó para alinearla con otros estándares internacionales, tales como *ITU-T V.24*, *V.28* e *ISO/IEC 2110*. [3].

En el caso del estándar *EIA/TIA-422* o RS-422 fue introducido en 1975 con el objetivo de permitir la transferencia a más altas velocidades a través de líneas de datos serie, logrando más distancia de lo que era posible con RS-232 y permitiendo múltiples receptores para un transmisor, para todo ello se utilizan 2 pares de hilos y señales diferenciales. La última revisión del estándar –la revisión B- fue publicada en 1994 y posteriormente fue reafirmada en el año 2005, igualmente también se encuentra alineada con su equivalente internacional, la recomendación *ITU-T-REC-V.11* también conocida como *X.27.* [4]

Cabe destacar que posteriormente a estos estándares se publicó el estándar *EIA/TIA-485* [5], y aunque inicialmente se encuentra fuera del alcance de este trabajo fin de grado, constituyó una evolución técnica sobre RS-422, permitiendo poder usar un solo par de hilos con múltiples transmisores y

receptores. RS-485, junto con sus antecesores objeto de este estudio, también es ampliamente utilizado en el ámbito industrial y de las comunicaciones serie.

Hoy en día, en el ámbito de estas comunicaciones, existen multitud de programas que son capaces de enviar o recibir datos por los puertos serie de un computador. Algunos de ellos son muy conocidos y útiles, tales como *PuTTY*, *Hyperterminal o Realterm*, pero con la progresiva desaparición de los puertos serie físicos en los PC´s y la aparición de convertidores USB a serie -que crean un puerto virtual- ha resultado en que estas capacidades se hayan visto limitadas. Esto ha sido debido principalmente a que estos programas no suelen ser capaces de controlar el sistema operativo o el hardware convertidor a un nivel lo suficientemente bajo, como para obtener resultados precisos y deterministas, sobre todo en el dominio del tiempo. [6].

En cambio, el análisis de los datos en PC´s ha evolucionado mucho más, existiendo múltiples programas de recepción y análisis de protocolos e incluso analizadores lógicos basados en hardware [7,8], pero prácticamente todos están centrados en la recepción de los datos y su posterior análisis, no siendo fácil encontrar dispositivos en el mercado que sean capaces de reproducir y transmitir estos datos por un puerto serie -con las mismas características que cuando se han recibido-, sin depender de los procesos del sistema operativo y en los que casi siempre se emplea en ello un dispositivo externo que tiene que convertir del puerto USB a RS-232 o RS-422 y que afecta de alguna manera a la comunicación.

2.2. Sistemas disponibles en el mercado

Dentro de los sistemas disponibles que podrían ser de utilidad para el propósito de este proyecto podemos distinguir entre convertidores USB a serie, software de simulación y dispositivos integrados por hardware/software.

2.2.1. Convertidores USB a Serie

Profundizando en los convertidores de puerto USB a RS-232 y RS-422, existen comercialmente muchos modelos de diferentes fabricantes. Dadas las

características de la comunicación USB, se requiere de *drivers* de dispositivo para interactuar con los sistemas operativos modernos, y esto -si cabe- dificulta aún más la utilización de estos dispositivos con las diferentes arquitecturas de microprocesadores existentes en el mercado.[6,9]

Entre los convertidores de USB a serie más populares existen los basados en el circuito integrado PL2303 del fabricante *Prolific*. [10]

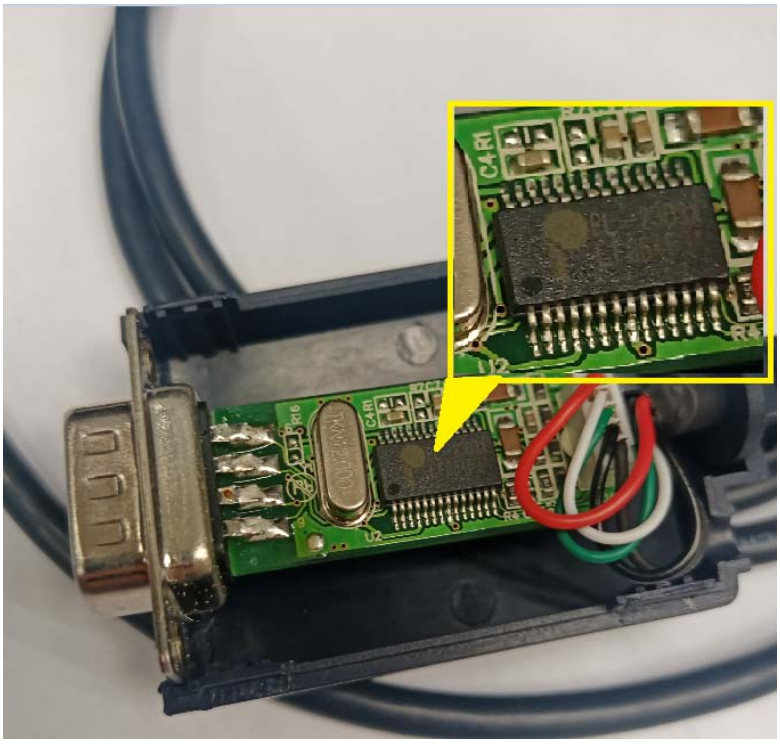


Figura 3: Circuito PL2303 en un conversor USB comercial

Este circuito integrado hace de interfaz entre un puerto USB y las señales TTL asíncronas para manejar un puerto serie, disponiendo en un solo chip de todos los módulos necesarios para realizar esta labor.

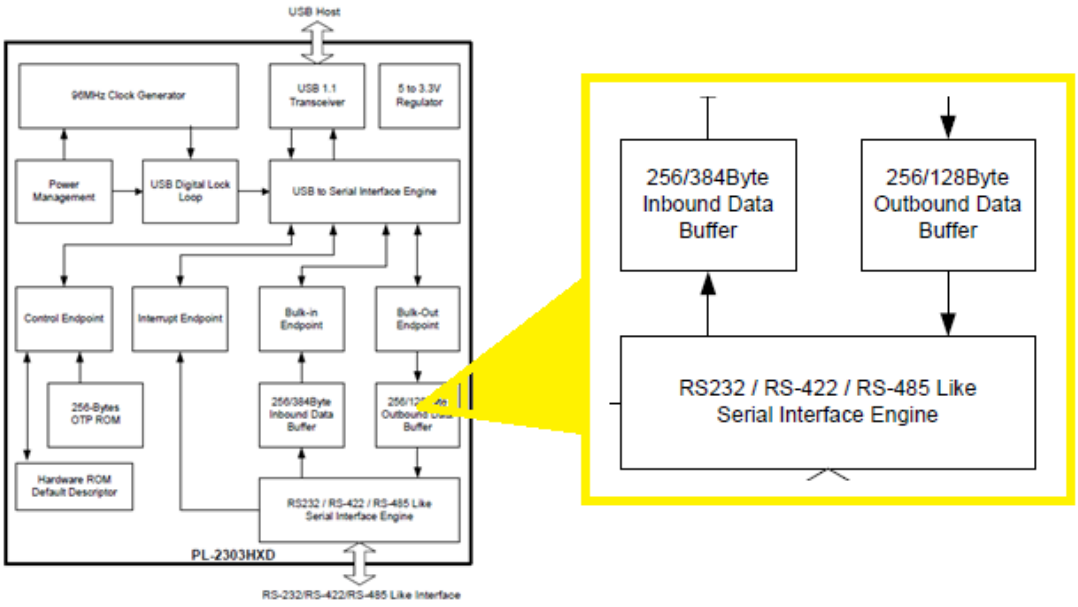


Figura 4: Esquema de bloques PL-2303

En la figura 4 se puede apreciar -entre otros módulos- como el *PL2303* incluye sendos *buffers* de entrada-salida de 256/128 Bytes, estos harán de pila FIFO a la hora de manejar los diferentes tiempos que pueden tener protocolos tan diferentes como el USB y el RS-232.

Dada estas diferencias y el encolado de mensajes -que hoy en día pueden llegar a tamaños de 1024 bytes/mensaje-, se puede dar la circunstancia de que los tiempos de separación entre mensajes no sea estrictamente reproducida, debido a la carga de trabajo de CPU, interrupciones, prioridades de procesos, etc.

Table 7-2 Baud Rate Settings (Supported by Driver)

Baud Rates (bps)	Baud Rates (bps)	Baud Rates (bps)	Baud Rates (bps)	Baud Rates (bps)
12000000				
6000000	460800	134400	19200	1800
3000000	403200	128000	14400	1200
2457600	268800	115200	9600	600
1228800	256000	57600	7200	300
921600	230400	56000	4800	150
806400	201600	38400	3600	110
614400	161280	28800	2400	75

Note: For special baud rate requirements, please contact Prolific FAE for driver customization support.

Figura 5: Baud Rates del IC PL-2303

El chip *PL2303* también dispone de un registro para programar entre otros el ratio de bits o *baudrate*, este registro permite configurar velocidades pre-establecidas en una tabla (figura 5), lo que imposibilita una libre elección de otras velocidades fuera de ella, siendo esto una limitación que solo pude ser solventada mediante una personalización *ad-hoc* del producto por parte del fabricante.

Aunque en la hoja de datos se declaran velocidades de hasta 12 Mbps no hay que olvidar que este circuito a su salida ofrece señales con niveles de tensión TTL que podrían ser manejados por la UART de un microcontrolador, pero que deben ser convertidas a los niveles adecuados especificados por las normas *TIA/EIA* correspondientes a RS-232 y RS-422, requiriendo un circuito apropiado para realizar esta función.

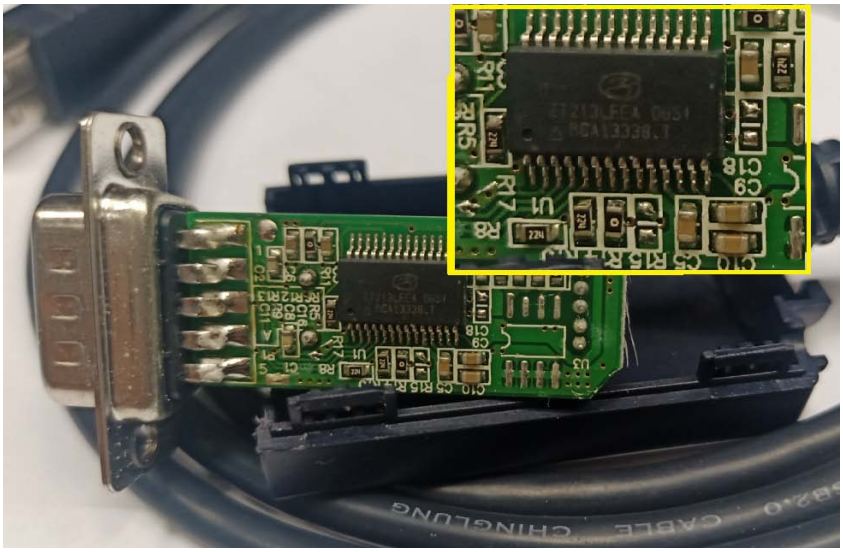


Figura 6: Circuito ZT213 en un conversor USB comercial

El driver comúnmente empleado para esta labor de conversión de niveles TTL-RS-232 suele ser del tipo *ZT213* [11] como el mostrado en la figura 6 o del tipo *MAX3232* [12] entre otros y que según sus hojas de datos solo garantizan una velocidad máxima de 250 Kbps, por lo tanto no permitirían generar velocidades superiores con garantías. Esto sin duda reduce el número de convertidores USB disponibles en el mercado a un selecto grupo que disponen de otro tipo de drivers que puede garantizar velocidades superiores y que muchas veces son de coste sensiblemente superior o de difícil disponibilidad.

Otro de los *chipset* disponibles en el mercado para la conversión de USB a TTL y también ampliamente utilizado en conversores es el modelo *CP2102* [13] de *SiliconLabs*, este circuito se integra ampliamente en estos dispositivos y dispone de un esquema de bloques similar al del *PL2302* y que se muestra en la figura 7.

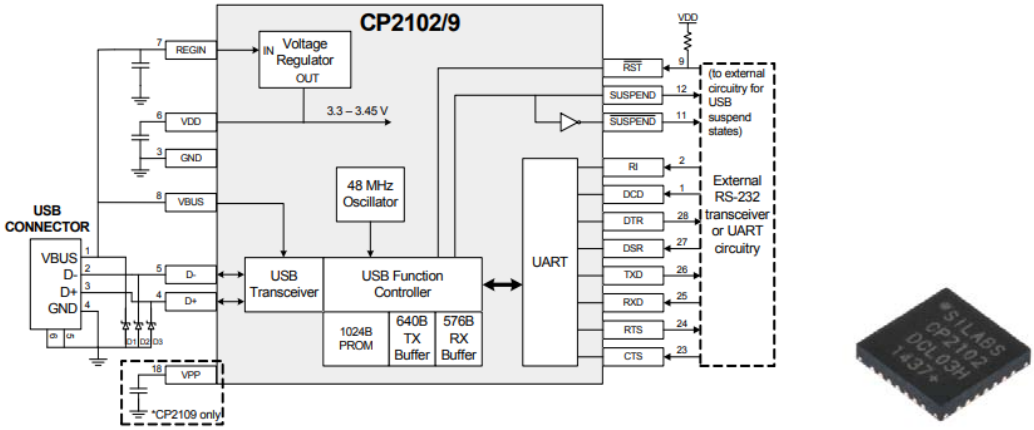


Figura 7: Bloques del circuito CP2102

De su hoja de datos también se desprende que comparte muchas características comunes con su competidor, siendo configurado también mediante una tabla de velocidades prefijadas, aunque en este caso solo es capaz de ser configurado hasta 921000 bps, pero en cambio dispone de un buffer de entrada-salida de 512 Bytes -que si bien es superior- también parece susceptible de ser llenado dada la longitud de mensajes utilizados hoy en día.

Igualmente al caso anterior, este circuito requiere de un driver convertidor de niveles TTL a RS-232 por lo que empleando los chips anteriormente citados -*ZL213* o *MAX3232*- se vería igualmente limitada la garantía de velocidad máxima a los 250 Kbps.

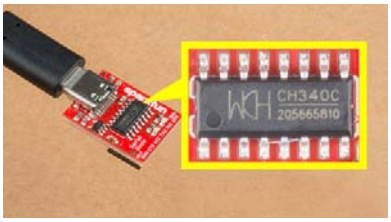


Figura 8: Circuito CH340 en conversor USB

Por último analizar el circuito integrado *CH340* del fabricante *WCH* [14], que al igual que los anteriores se integra en muchos convertidores USB de bajo coste y que realiza las mismas funciones con las mismas limitaciones. También incluye configuraciones de *baudrate* entre unos valores limitados y que alcanza un máximo de 2Mbps en la salida TTL, pero en este caso los convertidores TTL a RS-232 propuestos en la hoja de datos son de la serie *MAX213* y *MAX232,* que en ambos casos están más limitados aun que los anteriores en la velocidad garantizada, pues declaran *baudrates* máximos de 120Kbps. [15]

Remarcar que todos los circuitos convertidores estudiados son usados satisfactoriamente en muchas aplicaciones en las que es necesario recibir y transmitir datos en equipos que solo disponen de puertos USB y en las aplicaciones generales cumplen con los requisitos funcionales fundamentales, si bien de manera global no alcanzan los mínimos de velocidad o determinismo que se están buscando para este proyecto y en algunas otras aplicaciones más específicas.

2.2.2. Software de simulación

En cuanto al software de utilidad y como se ha introducido anteriormente, en el mercado existen varias aplicaciones capaces de interactuar con los puertos serie virtuales o físicos de los que dispone un ordenador. Algunas de ellas están orientadas a conexiones multiprotocolo, al estilo de la pantalla de terminal y pueden ser utilizadas para conexiones tanto RS-232 como otras tipo Telnet, FTP, SSH y demás, pues soportan diferentes modos de conexión para distintas aplicaciones.

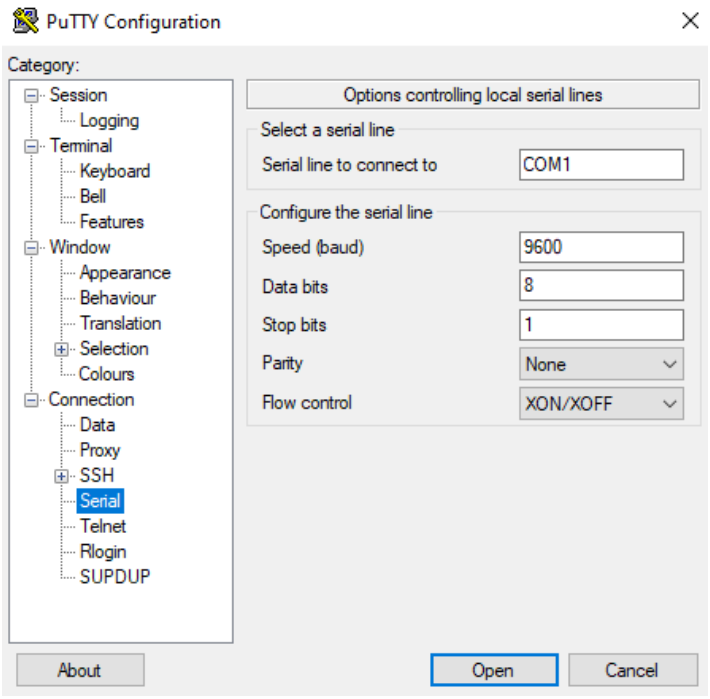


Figura 9: Pantalla de comunicación serie en PuTTY.

En el caso de *PuTTY* [16], es una aplicación gratuita ampliamente utilizada y que es capaz de conectarse al interlocutor seleccionando el puerto y los parámetros de la comunicación. Una vez conectado es posible enviar comandos introducidos manualmente en la pantalla de terminal e igualmente recibir los datos del puerto y mostrarlos por pantalla, pero dispone de escasas funciones para repetir o reproducir patrones de mensajes, hacer bucles etc. y su operación principal parece dirigida a los comandos introducidos por un operador de consola.

Como programas más evolucionados en las conexiones serie también podemos encontrar de manera gratuita los programas *Realterm* [17], *YAT* [18] *o Coolterm* [19] entre otros, que disponen de amplias funcionalidades entre las que se encuentran configurar el puerto con todas las posibilidades de bits, *baudrates* o paridades, enviar y recibir los datos en distintas codificaciones, realizar envíos cíclicos o con repeticiones, retardos y posibilidad de realizar automatizaciones.

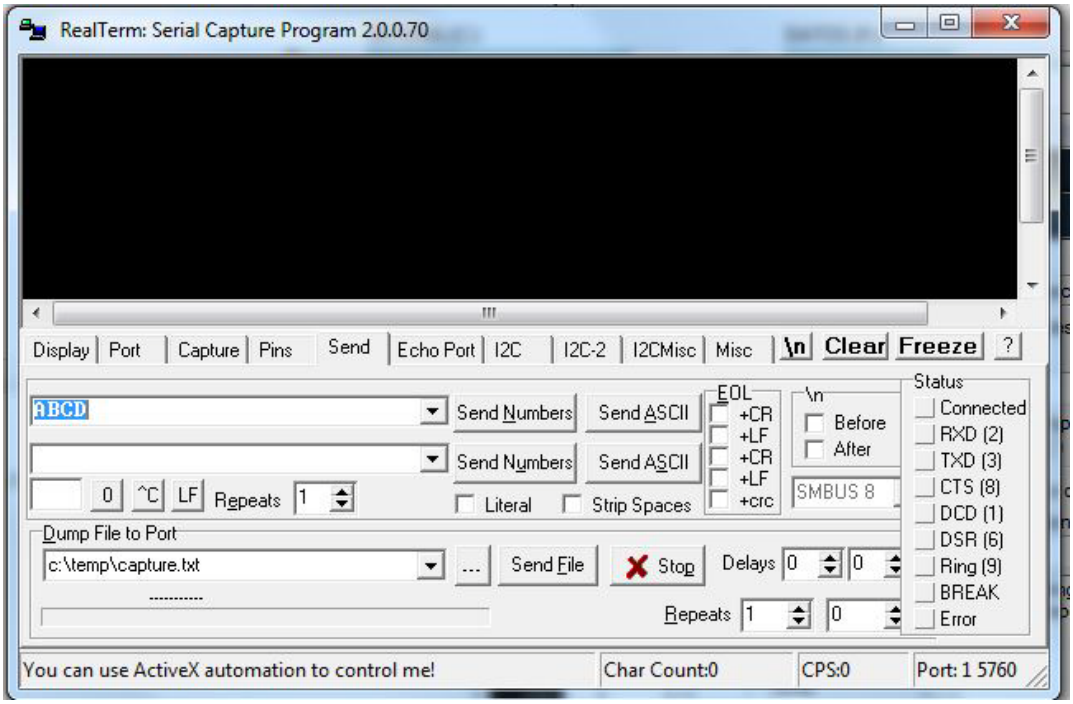


Figura 10: Pantalla de envio de *Realterm*

Además de los programas citados existen muchas otras aplicaciones -tanto gratuitas como comerciales- que ofrecen estas y otras funcionalidades, pero normalmente no van asociadas a un hardware de puerto serie específico, por lo que el control del puerto a través de los sistemas operativos multiproceso, junto con la diversidad de convertidores de USB a serie disponibles en el mercado hace que se pongan de manifiesto las limitaciones de velocidad y determinismo comentadas anteriormente.

2.2.3. Sistemas integrados Hw/Sw

Si buscamos sistemas integrados hardware/software que pudieran desempeñar las labores requeridas, existe en el mercado la plataforma *CompactRIO* del fabricante *National Instruments* [20]. Esta plataforma es un sistema embebido que combina un controlador en tiempo real con un módulo de E/S y FPGA en un solo dispositivo compacto, modulable mediante tarjetas extraíbles. *CompactRIO* se utiliza comúnmente en aplicaciones de automatización industrial, monitorización y control de procesos y mediciones en entornos exigentes.

Figura 11: Sistema modular *CompactRIO* de *National Instruments*.

Los módulos *NI-9890* y *NI-9871* [21] son módulos de control de instrumentos, que integrados en la plataforma modular de *CompactRIO* permitirían controlar puertos de comunicaciones RS-232 y RS-422 hasta 1 Mbps, mediante la programación de una aplicación de control en su FPGA que realice las funciones deseadas.

El entorno de programación de *CompactRIO* proporciona herramientas y software de desarrollo para configurar, programar y depurar una aplicación de control utilizando estas tarjetas de puertos serie. Una de las características clave de *CompactRIO* es su capacidad de personalización y flexibilidad. Los usuarios/as pueden seleccionar los módulos de E/S adecuados para sus necesidades y programar el controlador en tiempo real utilizando lenguajes de programación como *LabVIEW*.

Con el sistema *CompactRIO* se podría desarrollar un equipo integrado que realizara las labores requeridas, pero nos encontramos ante varios inconvenientes, empezando por el coste, ya que solo los módulos de RS-232 y RS-422 superan los 1000 Euros cada uno, sin añadir el coste del chasis principal que también supera esa cifra y esto haría inviable el proyecto con el presupuesto disponible. Además requeriría de un desarrollo en *LabView* a medida junto con la dificultad para mostrar mensajes de configuración o estatus sin disponer de una pantalla conectada al chasis principal, por lo que también sería un dispositivo poco autónomo y no fácilmente portable.

3. Arquitectura del Sistema

3.1. Arquitectura de bloques

La arquitectura general del sistema está compuesta por dos partes diferenciadas, por un lado el dispositivo hardware que generará todas las señales eléctricas y procesos del sistema y por otro lado un software que consistirá en una aplicación ejecutable en PC, que servirá para configurar el dispositivo hardware con los requisitos de simulación establecidos. Una vez configurado y desconectado del PC, el hardware será capaz de simular las señales de forma independiente.

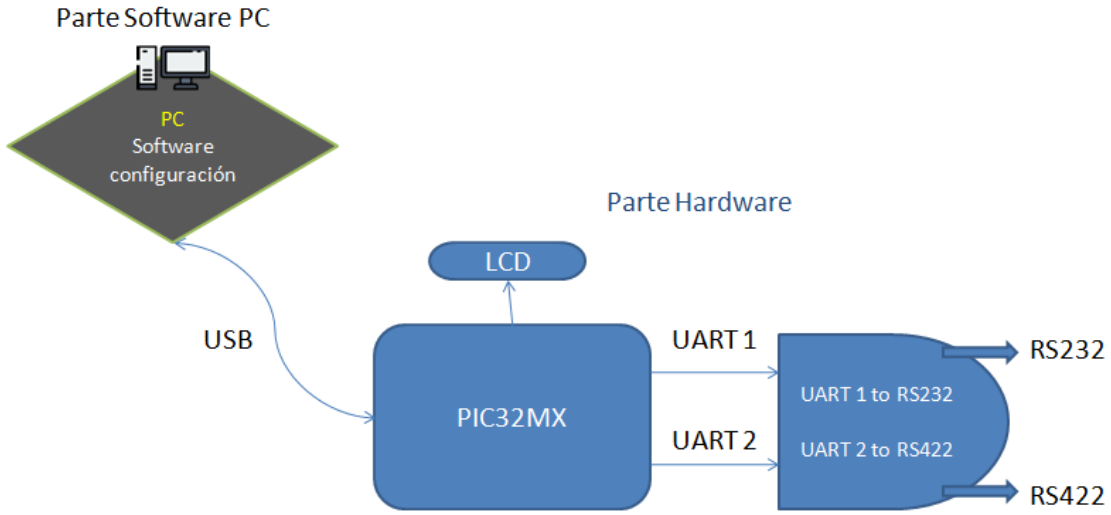


Figura 12: Esquema de bloques del sistema HW/SW

El hardware deberá generar señales RS-232 y RS-422 de hasta 1 Mbps, configurable en bits y paridad y capaz de enviar mensajes de hasta 1024 Bytes con capacidad de enviar los mensajes en bucle, con contador de mensajes y separación entre mensajes seleccionable en tiempo, igualmente dispondrá de una pantalla LCD donde mostrar configuración, mensajes de estado y error.

La aplicación de escritorio para PC deberá ser capaz de configurar todos estos parámetros y aplicarlos al hardware por USB con una interfaz gráfica de fácil uso.

3.2. Herramientas empleadas para desarrollo del Hardware

- Autodesk Eagle

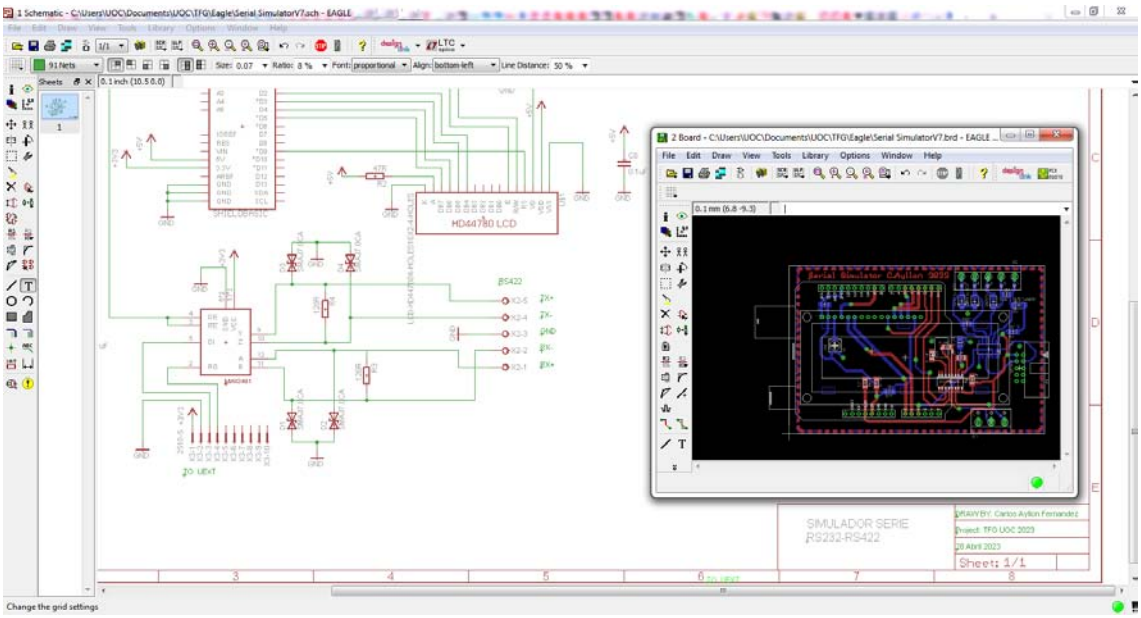


Figura 13: Imagen de Autodesk Eagle

Para el diseño electrónico y del circuito impreso se utilizará el software *Autodesk Eagle* [22]. Este software de diseño CAD/CAM dispone de todas las funcionalidades necesarias para diseñar los esquemáticos y realizar las rutas de pistas del circuito impreso, junto con otras funciones de gran utilidad, tales como la aplicación de reglas de diseño, depuración de errores, generación de listados, esquemas y producción de archivos en formato *gerber.* Además es capaz de usar gran variedad de componentes, en distintos encapsulados y formatos.

El uso de este software está muy extendido y se puede encontrar fácilmente librerías de componentes así como diseñar nuestras propias librerías, si no estuvieran disponibles, facilitando la incorporación de nuevos diseños o componentes personalizados.

- MPLAB® X Integrated Development Environment (IDE)

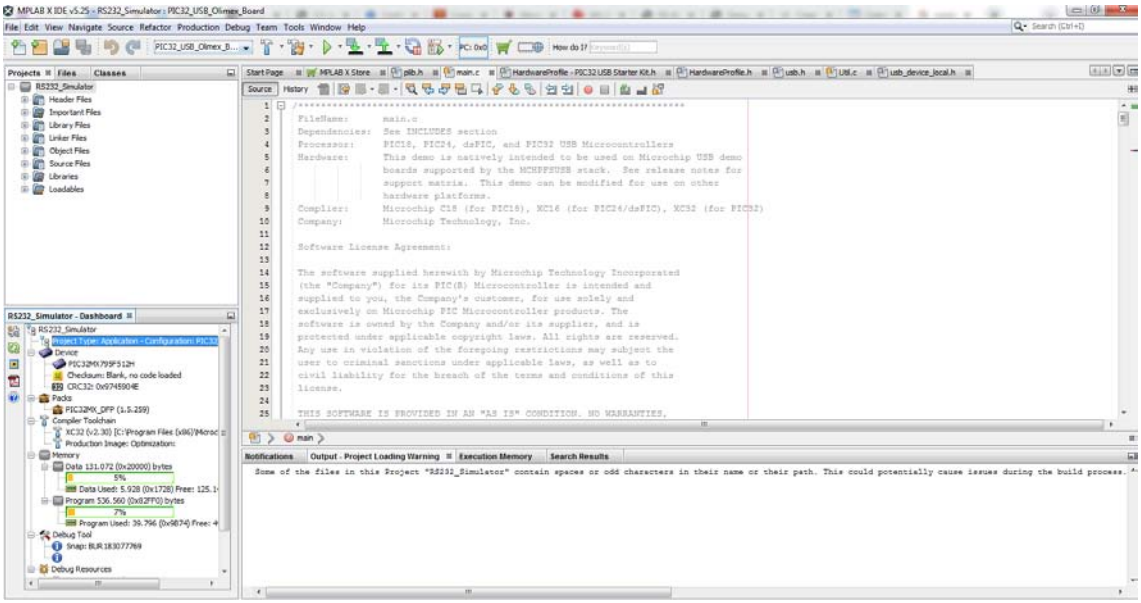


Figura 14: Imagen del IDE MPLAB X

Este entorno de desarrollo integrado -proporcionado por *Microchip Technologies*- [23] permitirá programar el microcontrolador PIC32 en lenguaje C, dispone de herramientas de compilación y depuración avanzadas junto con el soporte de variedad de librerías y ejemplos de uso facilitados por el fabricante. Igualmente se utilizará el compilador XC32 de Microchip que se integra perfectamente en el entorno de desarrollo y que esta optimizado para microcontroladores PIC de 32 bits.

- MPLAB® Snap In-Circuit Debugger/Programmer

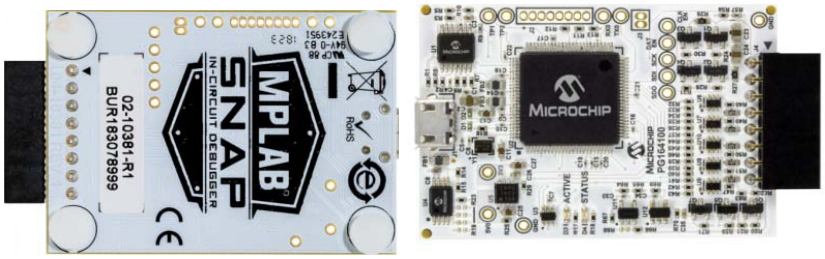


Figura 15: Programador MPLAB SNAP

Este dispositivo servirá para programar y depurar el *firmware* del microcontrolador. Se caracteriza por poder depurar en tiempo real, con puntos

de ruptura puede examinar direcciones de memoria, valor de variables y muchas otras posibilidades de bajo nivel que hacen posible la construcción de un programa eficiente y libre de errores.[24]

Dentro de los programadores disponibles por *Microchip* destaca por su modernidad, su bajo coste de adquisición y por ser capaz de soportar gran cantidad de dispositivos -no solo dispositivos PIC- sino también de las familias AVR y SAM, por lo que es una buena herramienta de laboratorio válida para este y otros proyectos.

- Otros recursos

Además de estos programas y útiles, para el montaje del prototipo se requerirán algunas herramientas de uso general, de soldadura, montaje etc. Para las comprobaciones y pruebas funcionales se requerirá el apoyo de otro tipo de útiles e instrumentación, tales como un programa de terminal, un polímetro, una fuente de alimentación y un osciloscopio o analizador de protocolos que permitirán comprobar fehacientemente el buen funcionamiento del prototipo para el cumplimiento de los requisitos de diseño establecidos.

3.3. Herramientas empleadas para desarrollo del Software

Como herramienta principal para el desarrollo de la aplicación de PC que configure el simulador se ha seleccionado el entorno de desarrollo integrado *Processing*.[25]

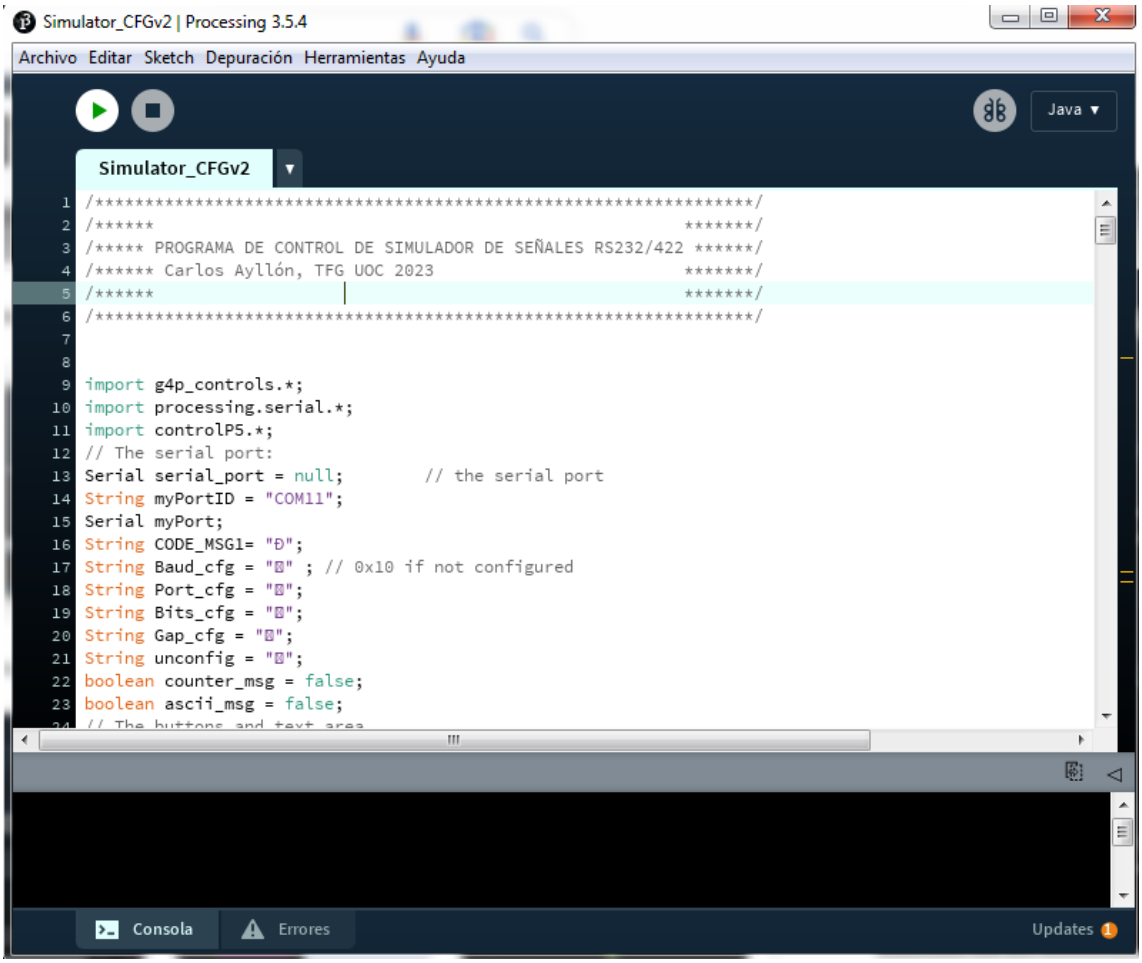


Figura 16: Imagen de *Processing IDE*

Processing es un lenguaje de programación y entorno de desarrollo integrado de código abierto GNU GPL, basado en Java y que es de muy fácil utilización. Al estar basado en este popular lenguaje hereda toda su funcionalidad, aunque hace uso de elementos de programación sencillos, por lo que no se requieren necesariamente conocimientos muy profundos en Java.

Processing proporciona una biblioteca gráfica incorporada que permite crear y manipular gráficos e imágenes de manera fácil. Esta biblioteca ofrece una amplia gama de funciones para dibujar formas, botones, aplicar colores y texturas y manejar eventos de entrada y salida entre otros, lo que facilita realizar una aplicación de escritorio en un entorno de rápido aprendizaje, además permite la importación de librerías externas para ampliar su funcionalidad. Esto permite aprovechar características adicionales, como la comunicación con hardware externo.

Este lenguaje además es multiplataforma, siendo compatible con *Windows*, *macOS* y *Linux*, lo que permite desarrollar proyectos en diferentes sistemas operativos. Para este proyecto en concreto permitirá su ejecución y la configuración del hardware en cualquier PC.

3.4. Estudio de placas de desarrollo de microcontrolador PIC

En el mercado se pueden encontrar diversas placas de evaluación o desarrollo sobre las que realizar un proyecto con microcontrolador. Para este trabajo, se ha seleccionado la familia de microcontroladores PIC32MX de *Microchip Technologies* [26], considerando que este microcontrolador ofrece una versatilidad muy amplia en toda su gama, disponiendo de unas características muy flexibles, rendimiento alto y unos recursos disponibles más que suficientes con costes muy reducidos.

Algunas de sus características fundamentales incluyen:

- Arquitectura MIPS: El PIC32MX utiliza la arquitectura MIPS32, que ofrece un alto rendimiento y eficiencia energética.
- Periféricos integrados: El microcontrolador incluye una variedad de periféricos integrados, como convertidores analógico-digitales, *timers*, comunicación UART y USB, que lo hacen adecuado para una amplia gama de aplicaciones.
- Memoria y almacenamiento: El PIC32MX tiene una memoria Flash de hasta 512 KB y RAM de hasta 128 KB pudiendo particionarse la flash a modo de EEPROM, también admite una amplia gama de interfaces de almacenamiento.
- Bajo consumo de energía: El microcontrolador es altamente eficiente en términos de consumo de energía, lo que lo hace adecuado para aplicaciones que requieren baterías o energía limitada.
- Herramientas de desarrollo: Microchip ofrece una amplia gama de herramientas de desarrollo, como compiladores, depuradores y emuladores, para facilitar el desarrollo de aplicaciones basadas en PIC32MX.

- Rendimiento de alta velocidad: El PIC32MX tiene una velocidad de reloj de hasta 80 MHz, lo que le permite realizar tareas complejas en ciclos de reloj más cortos.
- Compatibilidad: El PIC32MX es compatible con una amplia gama de interfaces de comunicación, lo que permite una fácil integración con otros dispositivos.

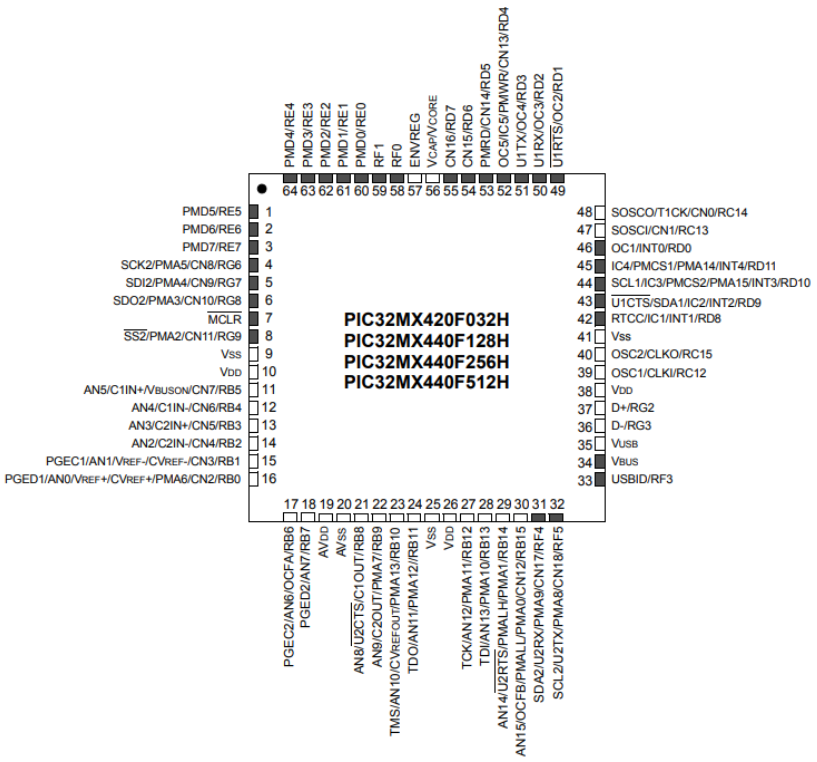


Figura 17: Detalle de patillaje del PIC32MX

Para este proyecto es de especial interés las características de los módulos UART [27]: de los que dispone esta familia, cuyas principales características son:

- Transmisión de datos *full-duplex*, de 8 o 9 bits.
- Opciones de paridad par, impar o sin paridad (para datos de 8 bits).
- Uno o dos bits de parada.
- Función de transmisión automática de hardware.
- Generador de tasa de bits totalmente integrado con *prescaler* de 16 bits.
- Tasas de baudios que van desde 76 bps a 20 Mbps a 80 MHz.
- Búferes de datos de recepción y transmisión separados (FIFO).

- Detección de errores de desbordamiento de búfer, tramas y paridad.
- Interrupciones de transmisión y recepción separadas.
- Modo de bucle invertido para soporte de diagnóstico.

Otra de las características a destacar de este microcontrolador es la memoria disponible y su flexibilidad para usarla virtualizada, que será especialmente útil en este proyecto, ya que se pretende usar la memoria flash a modo de memoria EEPROM no volátil, donde se almacenarán las configuraciones seleccionadas por el usuario/a.[28]

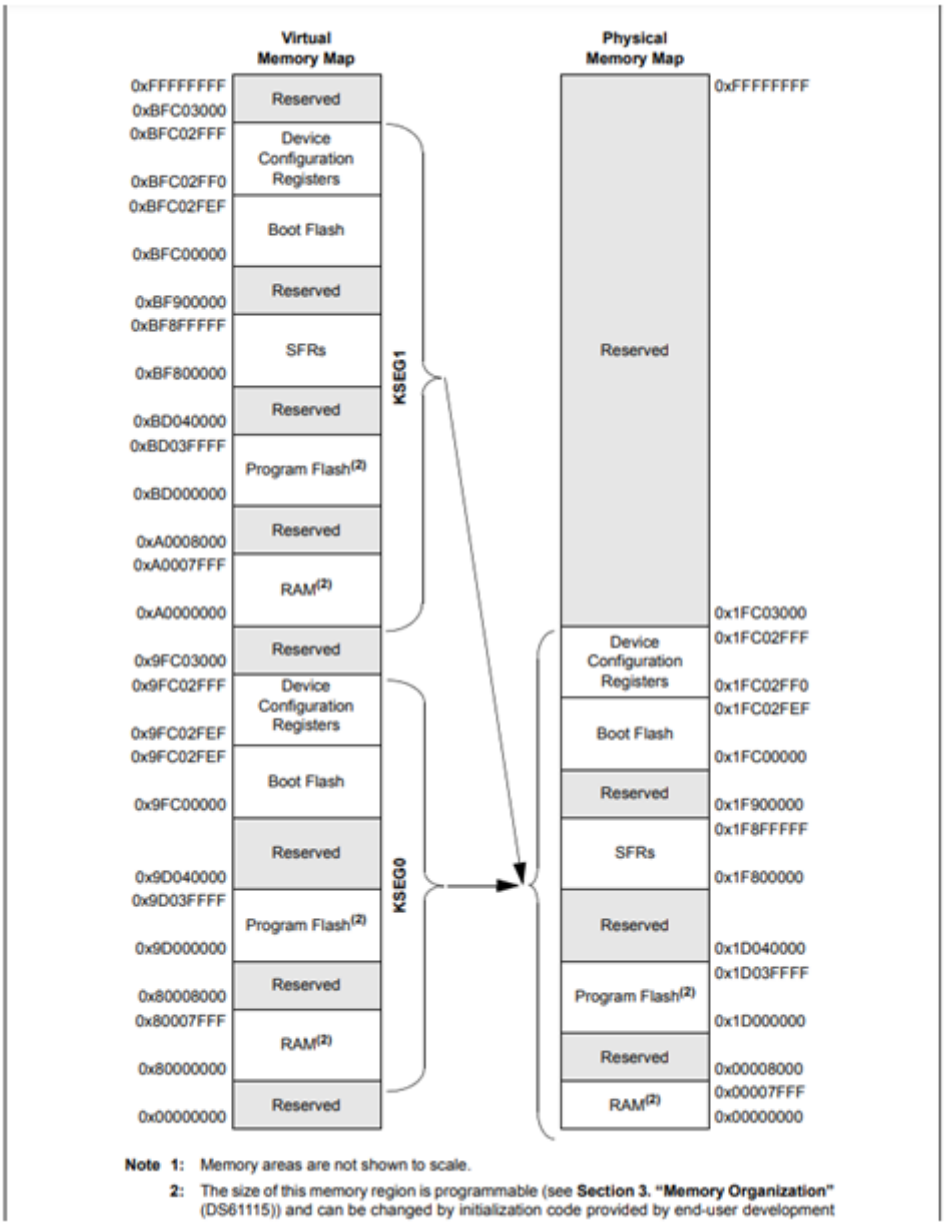


Figura 18: Organización de la memoria del PIC32MX

El puerto USB nativo del que dispone esta familia de microcontroladores también nos será de especial utilidad, pues se pretende poder configurar el dispositivo desde un ordenador personal y un software, que conectado por este puerto programe las diferentes configuraciones seleccionadas por el usuario/a.

Entre las placas de desarrollo disponibles que utilicen esta familia de microcontroladores se pueden encontrar tanto placas oficiales de Microchip como otras placas desarrolladas por otros fabricantes que también serían perfectamente válidas para este proyecto. A continuación se ofrece un pequeño resumen de características de una cuidada selección de las placas más apropiadas:

- Curiosity PIC32MX470 Development Board [29]

Figura 19: Curiosity PIC32MX Development Board

- PIC32MX470F512H 32-bit.
- Dos zócalos de expansión mikroBUS
- Dos pulsadores
- Conectores X32 para audio I/O
- Conector de expansión GPIO.
- Conector USB de programación/depuración
- Conector USB para conectividad con PIC32 USB (Device/Host mode) .
- Conector para 5V externos
- Coste aproximado a la publicación de este trabajo: 40€

- PIC32 USB STARTER KIT III [30]

Figura 20: Placa PIC32 USB STARTER KIT III

- PIC32MX450/470 MCU
- Programador/depurador integrado
- Dispositivo USB Host, Dual Role & OTG
- Conector a diversas placas de expansión
- Coste aproximado a la publicación de este trabajo: 83€

- OLIMEX PIC32-PINGUINO [31]

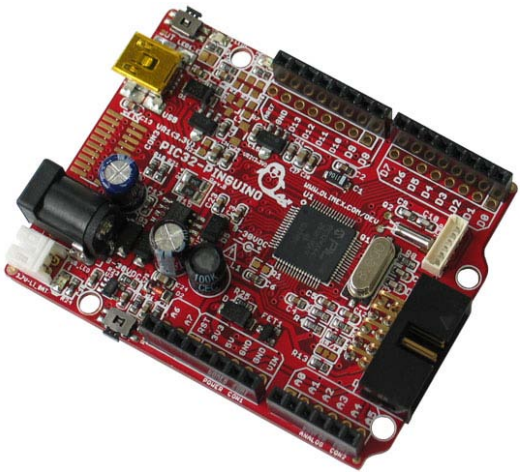


Figura 21: Placa PIC32-Pinguino de Olimex

- PIC32MX440F256H 80 MHz de 256KB Flash 32KB RAM
- Fuente DC-DC que permite alimentadores entre 9 a 30V
- Conector USB que permite alimentar la placa
- Conector ISP que permite usar programadores Microchip
- Conectores compatibles con shields de Arduino
- Coste aproximado a la publicación de este trabajo: 14€

- CLICKER 2 FOR PIC32 [32]

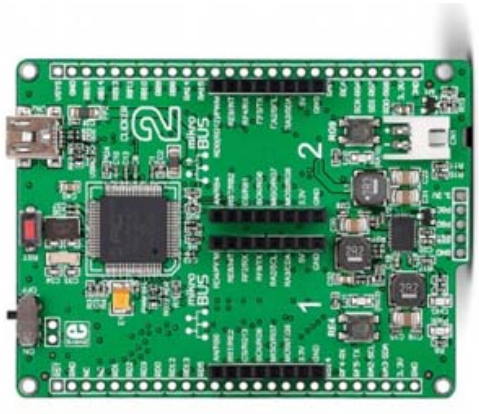


Figura 22: Placa CLICKER 2 FOR PIC32

- PIC32MX460F512L
- 80MHz/105DMIPS de operación
- 512+12KB de memoria de programa
- 52 GPIOs programables
- 2 zócalos mikroBUS
- Conector USB Mini-B
- Coste aproximado a la publicación de este trabajo: 53€

De todas las placas de desarrollo seleccionadas no hay un determinante técnico de exclusión, pues todas comparten los micros de la serie PIC32MX que disponen de los recursos requeridos suficientes para este proyecto, tales como frecuencia de funcionamiento, numero de puertos UART, USB, GPIO disponibles, etc. Por lo tanto -después de este primer filtro- y dado el limitado presupuesto económico del proyecto se ha decidido hacer del coste el principal elemento de selección, eligiendo por ello la placa PIC32-Pinguino como la más apropiada y asequible para este trabajo. Además su compatibilidad de forma y pines con las conexiones de *Arduino* puede facilitar el uso de interfaces o futuros diseños con esta misma placa, dada la gran difusión que ha alcanzado este tipo de módulos en los últimos años.

Cabe destacar que existen en el mercado algunas otras placas de desarrollo de PIC32 disponibles, tales como *ChipKit Fubarino* [33], *NU32* [34], pero que por

motivos de coste, características mínimas o dificultad de adquisición han sido previamente descartadas del análisis selectivo, aunque como se ha comentado el diseño podría ser adaptado a cualquier placa con un micro similar solo reubicando el diseño y los pines a la placa de desarrollo en cuestión.

3.5. Selección de componentes

Una vez seleccionada la placa PIC32-Pinguino, para diseñar un circuito que sirva de interface entre el microcontrolador y los puertos de comunicaciones RS232 y RS422 se deben seleccionar algunos componentes que deben cumplir con las especificaciones que se han requerido.

3.5.1. Circuitos Integrados:

El mayor punto crítico se encuentra en los circuitos integrados controladores de línea, que deben convertir los niveles de señal usados por los puertos del microcontrolador, -habitualmente entre 0V y +3V3 o TTL- a los niveles especificados en los estándares TIA/EIA-232 [1] y TIA/EIA-422 [2] que definen -en el caso del RS232- tensiones de +3V a +15V para un cero lógico, y -3V a -15V para el uno lógico, estableciéndose la región de transición entre +3V y -3V. De igual manera en circuito abierto no se deben sobrepasar los 25V de magnitud.

Para cumplir estos y otros requisitos de los estándares existen en el mercado variedad de circuitos integrados de diferentes características y encapsulados, algunos ampliamente utilizados y de bajo coste, como el MAX3232 de Texas Instruments y otros fabricantes. Este circuito está compuesto por 2 drivers y 2 receptores, compatibles tanto en alimentación como en señales con lógicas de 3V3 y 5V y que efectuaría bien esta labor, pero que se encuentra limitado en cuanto al ratio de bits operacional, estableciéndose en la hoja de datos del fabricante hasta un máximo de 250Kbps [35]. Este ratio de bits podría ser ampliamente válido en muchas aplicaciones, pero no cumple con los requisitos de este proyecto, pues como ya se ha descrito anteriormente se pretenden simular señales de hasta 1 Mbps.

Alternativamente el fabricante Texas Instruments en la hoja de datos del IC MAX3232 ya ofrece circuitos compatibles y de alta velocidad, como el SN65C3232 y el SN75C3232, este último con rango de temperatura entre 0ºC y 70ºC será el seleccionado para el propósito requerido, pues cumple ampliamente con los requisitos de diseño establecidos, garantizando 1 Mbps.[36]

En el caso de la conversión de niveles lógicos al par diferencial RS-422 ocurre algo similar, con niveles lógicos entre -10V y +10V se dispone de varios dispositivos muy comunes en el mercado, como el circuito integrado MAX3483 fabricado por *Analog Devices* (antes *Maxim*), pero que solo garantiza ratios de datos de 250 Kbps y otros como el MAX3485, MAX3491 ó MAX3491 [37] que llegan hasta los 10Mbps garantizados. Siendo cualquiera de estos últimos de utilidad, se ha seleccionado el MAX3491, por disponer de características más flexibles en cuanto a activación del driver y capacidad *full-dúplex*, que aunque no son estrictamente necesarias en este diseño sí que pudieran ser útiles en futuras funcionalidades.

Selection Table

PART NUMBER	GUARANTEED DATA RATE (Mbps)	SUPPLY VOLTAGE (V)	HALF/FULL DUPLEX	SLEW-RATE LIMITED	DRIVER/ RECEIVER ENABLE	SHUTDOWN CURRENT (nA)	PIN COUNT
MAX3483	0.25	3.0 to 3.6	Half	Yes	Yes	2	8
MAX3485	10		Half	No	Yes	2	8
MAX3486	2.5		Half	Yes	Yes	2	8
MAX3488	0.25		Full	Yes	No	—	8
MAX3490	10		Full	No	No	—	8
MAX3491	10		Full	No	Yes	2	14

Figura 23: Tabla de selección de drivers TIA/EIA-422

3.5.2. Otros componentes:

Las líneas RS-422 son susceptibles de abarcar largas distancias, es por ello que deben ser protegidas contra transitorios u otras descargas que pueden ingresar al circuito a través del cableado, por ello se incorporarán al diseño unos diodos supresores del tipo *transil* SMAJ7.0CA [19] para prevenir daños al prototipo en caso de descargas. [38,39]

Otro componente a seleccionar es la pantalla LCD, donde se mostrarán mensajes de configuración, estados y otros. En el mercado se pueden encontrar varios modelos con diferentes interfaces, pero están ampliamente disponibles las pantallas LCD compatibles con el controlador Hitachi HD44780. Estas pantallas de cristal líquido se vienen suministrando en varios formatos por múltiples fabricantes y destacan especialmente por su reducido coste, compatibilidad, amplia documentación y reducido tamaño.

Figura 24: Pantalla LCD 1602

En este caso se ha seleccionado la versión de 2 filas de 16 caracteres LCD1602 con retroiluminación LED e interfaz paralelo que permite la conexión usando 4 u 8 líneas de datos.[40]

El resto de componentes necesarios serán componentes discretos de uso general, tales como resistencias, condensadores, conectores y un potenciómetro para la regulación del contraste del LCD. Para las salidas de las señales generadas se ha optado por unos bloques de terminales con tornillo, ya que al tratarse de una herramienta que puede usarse en diferentes aplicaciones se podría conectar en ellos desde un par de cables a cualquier tipo de latiguillo con conector.

En cuanto al circuito impreso se realizará el prototipo en una placa FR4 estándar que se adosará con conectores a modo de escudo a la placa de desarrollo del PIC32, aprovechando el factor de forma de *Arduino* que incluye el modelo PIC32-Pinguino elegido.

3.6. Diseño electrónico

3.6.1. Cálculos:

A la hora de enfrentarse al diseño electrónico se deben evaluar las características de los componentes y realizar los cálculos pertinentes para garantizar que todo el sistema va a trabajar dentro de los límites de funcionamiento esperados y especificados por los fabricantes de los circuitos integrados.

En cuanto a lo que a consumos se refiere, la placa de desarrollo de PIC32 debe de ser capaz de alimentar a todos los componentes de nuestro circuito y, si no fuera capaz, se haría necesario plantear en el diseño una fuente de alimentación adicional. Por ello se han calculado los consumos estimados del diseño aunando los consumos máximos de los componentes especificados en sus hojas de datos:

Componente	*Consumo*	*Alimentación*
MAX3491	2.2 mA	3V3
SN75C3232	1.0 mA	3V3
LCD 1602	39 mA	5V
Total suma:	3.3 mA	3V3
Total suma:	39 mA	5V

Tabla 3: Consumo de los componentes

Con estos valores estimados, y sin disponer de información adicional en la documentación de la placa, si revisamos la fuente de alimentación en los esquemas, vemos que ésta se compone de dos circuitos reguladores, uno para los 5V el *MC33063ADR* [41] y otro para los 3V3 el *MCP1700T-3302* [42]:

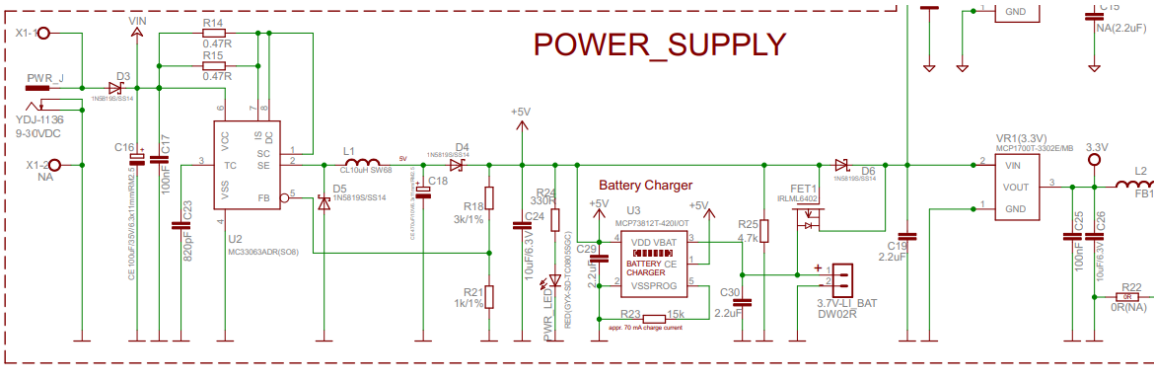


Figura 25: Fuente de alimentación PIC32-Pinguino.

Evaluando las hojas de datos de estos componentes se puede comprobar que la corriente máxima suministrada por estos reguladores es de 1.5A y 250mA para las tensiones de 5V y 3V3 respectivamente, por lo que se estima que los reguladores pueden proporcionar una corriente más que suficiente para nuestro diseño, que apenas consumirá en conjunto unos pocos miliamperios.

En cuanto al diseño del circuito impreso, con las corrientes máximas estimadas no será necesario tener especial precaución en cuanto al área o tamaño de las pistas, pues si se realizan los cálculos se puede comprobar que con pistas de ancho inferior a 1mm se podrían ya manejar corrientes muy superiores a las estimadas de acuerdo a IPC-2221 [43]

3.6.2. Diseño de esquemáticos

Utilizando el software de diseño *Eagle* y evaluando en detalle las hojas de datos, tanto del microcontrolador PIC32MX, como de los circuitos integrados y demás componentes, se ha realizado el diseño del circuito cuyo esquemático se incluye completo en el Anexo I.

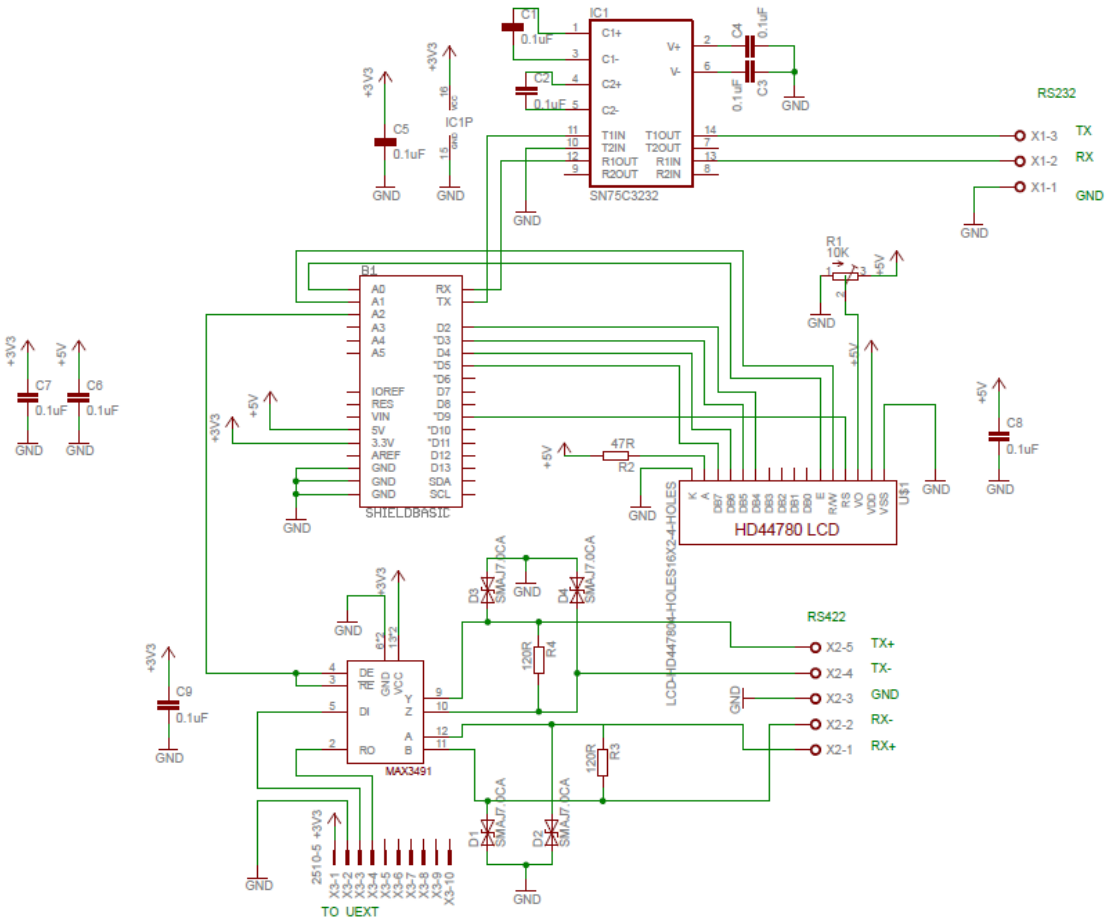


Figura 26: Esquema general del diseño.

El circuito se ha diseñado para conectar directamente con la placa de desarrollo de PIC32-Pinguino en las conexiones que tiene éste disponible y cuyo esquema se adjunta en Anexo V. Para ello se ha utilizado como componente de diseño la huella del *shield Arduino*, que se ensambla con conectores macho de paso 2.54mm a la parte correspondiente del PIC32-Pinguino. Se alimenta a través de éste con la fuente de la placa de desarrollo -como se ha calculado anteriormente- obteniendo los 3V3 para alimentar los circuitos integrados y los 5V para el display LCD.

El circuito recibe las conexiones de los puertos UART1 y UART2 del microcontrolador y a través de los circuitos integrados IC1 e IC2 convertirá los niveles lógicos de las señales generadas por las UART a los niveles admisibles dentro de los estándar TIA/EIA-232 y TIA/EIA-422. El display LCD se conecta en su mayoría con líneas paralelas por el puerto *D* del PIC32 excepto las señales *RW* y *E* que lo hacen a través del puerto *B.*

En la medida de lo posible se han seleccionado los componentes con encapsulados en formato SMD de soldadura superficial, para posteriormente reducir el número de taladros pasantes, facilitar el montaje y reducir el tamaño del circuito, que en gran medida vendrá determinado por las dimensiones del LCD y de la placa de desarrollo.

En el caso del driver SN75C3232 se han seguido las recomendaciones de la hoja de datos del fabricante, en cuanto a la selección de condensadores de la bomba de carga, teniendo en cuenta que la alimentación elegida para este circuito es de 3V3, se han colocado condensadores de 0.1uF.

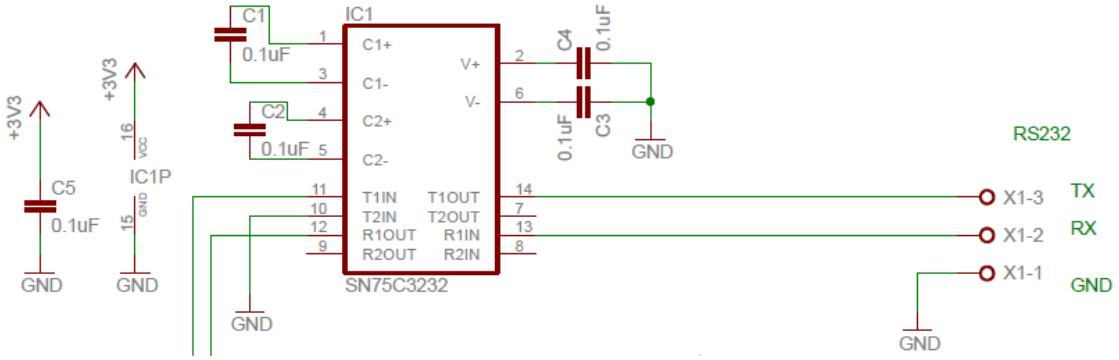


Figura 27: Esquema conversión RS-232.

En el caso de la salida RS422 y dado que los pines de la UART 2 del PIC32 no tienen conexión a esta zona del *shield*, sino que se encuentran disponibles en el llamado conector UEXT de la placa, se han tenido que conexionar estos pines a un conector -que en nuestro diseño hemos llamado X3- y que posteriormente conectaremos mediante un cable flexible y conectores del tipo IDC.

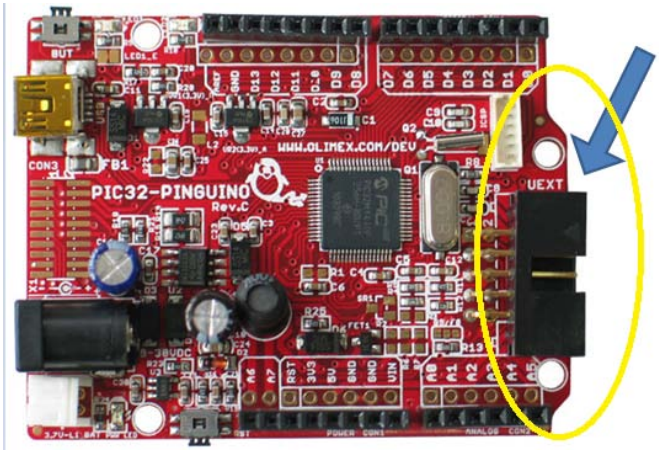


Figura 28: Detalle conector UEXT en placa PIC32-Pinguino.

Cabe destacar que -aunque no son necesarios en esta aplicación- se han conectado también entre sí los pines de entrada RX de los drivers y de las UART1 y 2, pues aunque este simulador solo pretende transmitir usando los pines TX, el dejar ya conectados lo pines de entrada RX abre la posibilidad a futuras aplicaciones o desarrollos con esta misma placa.

De igual manera en el conexionado del circuito MAX3491 se han conectado los pines de control de *driver enable* y *receiver enable* a un pin del microcontrolador, pues se ha considerado que en futuros desarrollos podría ser útil tener disponible esta característica de control del driver para otros usos.

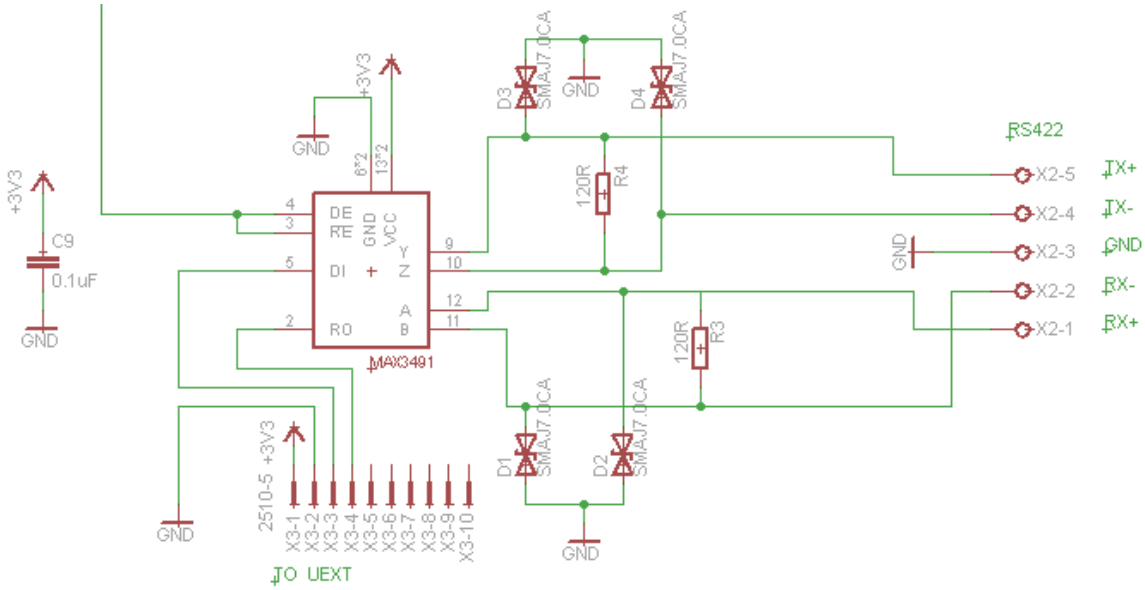


Figura 29: Esquema conversión RS-422.

El circuito diseñado incorpora los diodos supresores de transitorios -ya comentados en la selección de componentes- con el objeto de prevenir posibles daños por descargas. También se han incorporado las resistencias de carga de impedancia en la salida de RS-422 así como condensadores de filtro en las líneas de alimentación para evitar parásitos y fluctuaciones.

3.6.3. Diseño del circuito impreso o PCB

Se ha optado por realizar un diseño del circuito impreso -detallado en el Anexo II- con pistas a doble cara, para así reducir en lo posible el tamaño, optimizando el espacio, y como se ha comentado antes, empleando el mayor número de componentes SMD posible para evitar taladros y reducir las dimensiones.

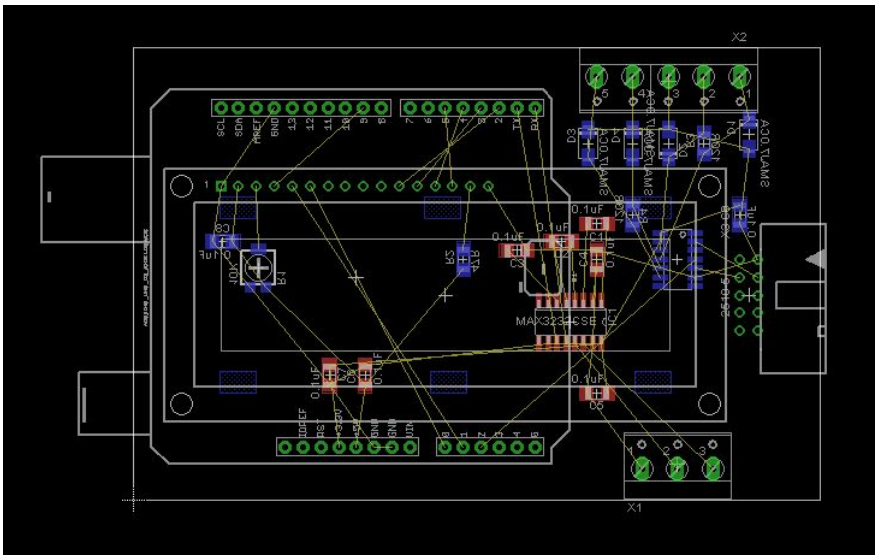


Figura 30: Disposición en PCB previo al enrutado

Dado que la placa del prototipo se va a fabricar con métodos tradicionales y no industrialmente se ha optado por tomar ciertas reglas de diseño, ya que por ejemplo las dimensiones de las pistas, el número de vías o los taladros metalizados influirán sensiblemente en la fabricación, ya que es difícil realizar el metalizado de taladros o vías sin disponer de métodos industriales.

Igualmente y dado que la soldadura de los componentes se realizará de forma manual, se ha tratado que los propios componentes no obstaculicen la

soldadura, por ejemplo: todas los pistas hacia los pines del display LCD se han enrutado por la capa inferior, pues si se enrutaran por la capa superior el propio encapsulado del display no permitiría soldarlo, y al no estar los taladros metalizados, no tendría conexión eléctrica. Igualmente ocurre con los conectores de salida X1 y X2, comprobando todo por partida doble con el objeto de mitigar en lo posible los riesgos establecidos en la tabla 1.

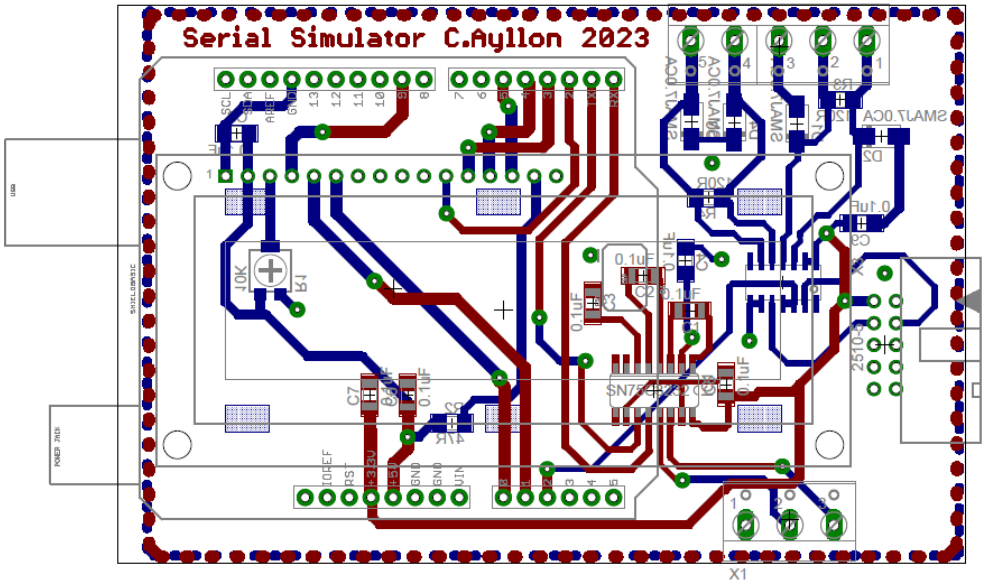


Figura 31: Diseño final de la PCB

Estas licencias que se han tomado a la hora de diseñar el circuito sin duda podrían haberse obviado si el circuito impreso se fuera a producir con métodos industriales, resultando en un diseño más limpio y reducido, ya que la fabricación manual impide realizar un diseño 100% idóneo y que incluya máscaras, serigrafía, taladros metalizados, pistas finas, etc.

Aún con las limitaciones que se han tenido que aplicar para hacer viable la producción del prototipo sin enviarlo a fabricar por métodos industriales, se ha tratado de realizar el trazado de las pistas siguiendo las mejores prácticas, de cara a evitar la posibilidad de interferencia electromagnética, teniendo en cuenta que las señales de comunicación a altas velocidades son susceptibles de causar este tipo de problemas [44]. Dentro de lo posible también se ha tratado de seguir las recomendaciones de *layout* de las hojas de datos de los fabricantes y procurando proveer planos de GND para evitar interferencias.

3.6.4. Fabricación y montaje

Una vez reproducido el diseño sobre una placa de cobre y taladrado convenientemente, se ha procedido a estañar la placa para evitar la oxidación y se ha realizado el montaje y soldadura de los componentes.

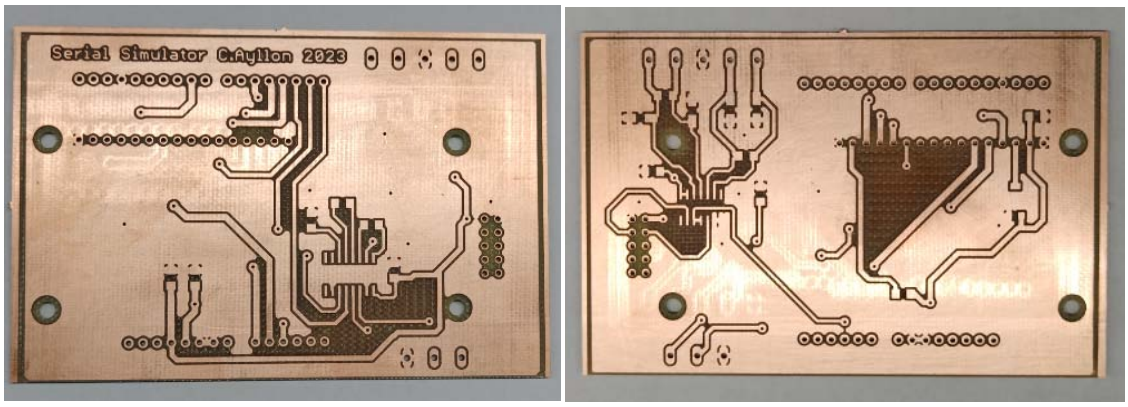


Figura 32: PCB por ambas caras previo al montaje

Finalizado el montaje de los componentes por ambas caras y antes de cualquier conexionado se ha realizado una minuciosa comprobación de todas las soldaduras y se ha verificado con el polímetro que no haya cortocircuitos o algún tipo de defecto o error que pudiera causar algún daño o mal funcionamiento, todo ello orientado a mitigar el riesgo R06, de acuerdo a la tabla 2 del plan de contingencias descrito en el capítulo 1.

Igualmente se ha fabricado el cable plano con conectores IDC que servirá de unión entre las dos placas para la transmisión de las señales de la UART2 al driver MAX3491.

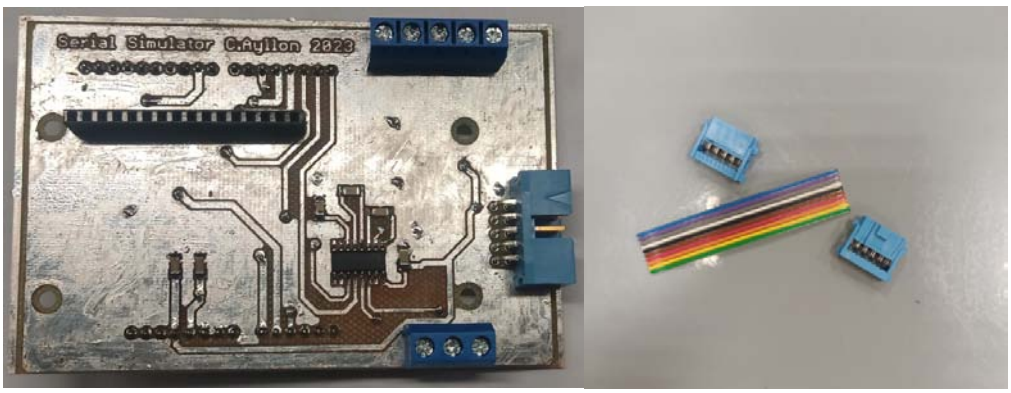


Figura 33: PCB: Montaje de componentes y conectores

3.6.5. Pruebas preliminares

Una vez montado el circuito por completo y tras la revisión de fabricación anterior, se ha alimentado externamente con una fuente de alimentación por los pines de 3V3 y 5V para comprobar que todos los puntos reciben dichas tensiones, midiendo las corrientes y comprobando que no hay consumos no previstos.

Una vez se ha conectado el módulo a la placa de desarrollo y comprobado que toda la parte mecánica y de dimensiones encaja y conecta según lo diseñado, se han realizado pequeñas porciones de código con MPLAB X IDE para comprobar que todas las conexiones son correctas y probar los componentes que vamos a usar por separado.

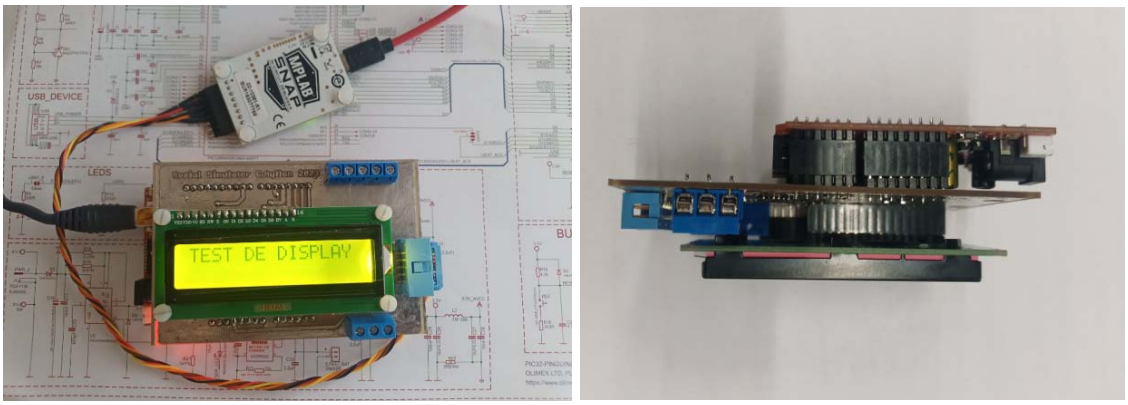


Figura 34: Ensamblaje y pruebas preliminares

3.7. Diseño del firmware del microcontrolador

Llegados al punto en el que disponemos de un hardware funcional para nuestro proyecto debemos diseñar y programar las funciones que deberá ejecutar nuestro sistema a través del microcontrolador. Toda esta programación la realizaremos con el entorno de desarrollo MPLAB X IDE, programando en lenguaje C junto con el compilador XC32.

3.7.1. Diagrama de estados

En la figura 35 se resumen de manera global los estados por lo que pasará el dispositivo en base al firmware programado, cuyo código se incluye en detalle en el Anexo IV:

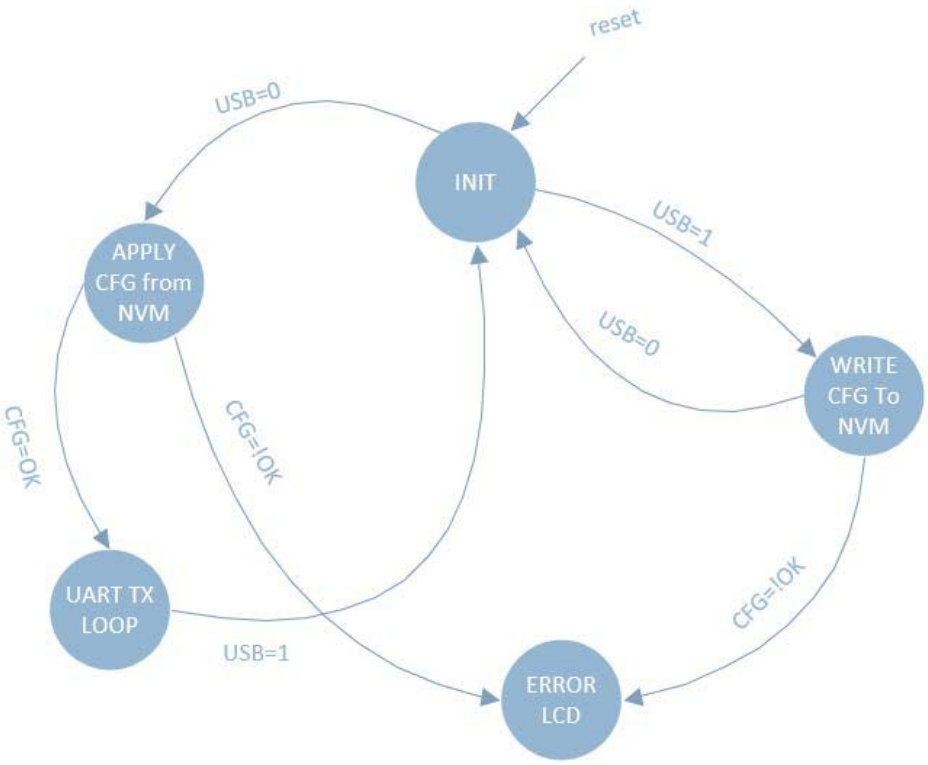


Figura 35: Diagrama de estados del firmware

El sistema define sus cambios básicamente por dos condiciones, la conexión al puerto USB -que puede estar conectado o no-, y la existencia o no de una configuración válida en la memoria no volátil o NVM.

Si al iniciar el sistema éste está desconectado del PC (USB=0) se buscará y aplicará la última configuración existente en la memoria no volátil (NVM), y si esta configuración existe y es válida (*CFG=OK*) entrará en el bucle de transmisión con ella y solo saldrá de allí por conexión al PC para configurarlo (USB=1). Si por el contrario la configuración no fuera válida, no entrará en el

loop de UART sino que pasará a mostrar la condición de error en el display LCD.

En el caso de estar el puerto USB conectado al PC (USB=1) el sistema entra en modo de configuración y esperará del PC una configuración válida que escribirá en la memoria no volátil, en este caso se escribirá y tan pronto se desconecte del PC (USB=0) comenzará con los estados del párrafo anterior. Si por el contrario la configuración recibida desde el PC no es válida se mostrará tal condición de error en la pantalla LCD.

3.7.2. Configuración del microcontrolador

- Oscilador del PIC32MX440F256H:

El oscilador del PIC32MX es altamente configurable. Las diferentes opciones de reloj le permiten maximizar el rendimiento del dispositivo mientras controla el consumo de energía en otras partes del micro. En nuestro caso el microcontrolador obtiene la señal de reloj de un oscilador primario proporcionado por un cuarzo de 8MHz y a partir de él se configuran diferentes registros para obtener las frecuencias deseadas con divisores o multiplicadores mediante PLL. [45]

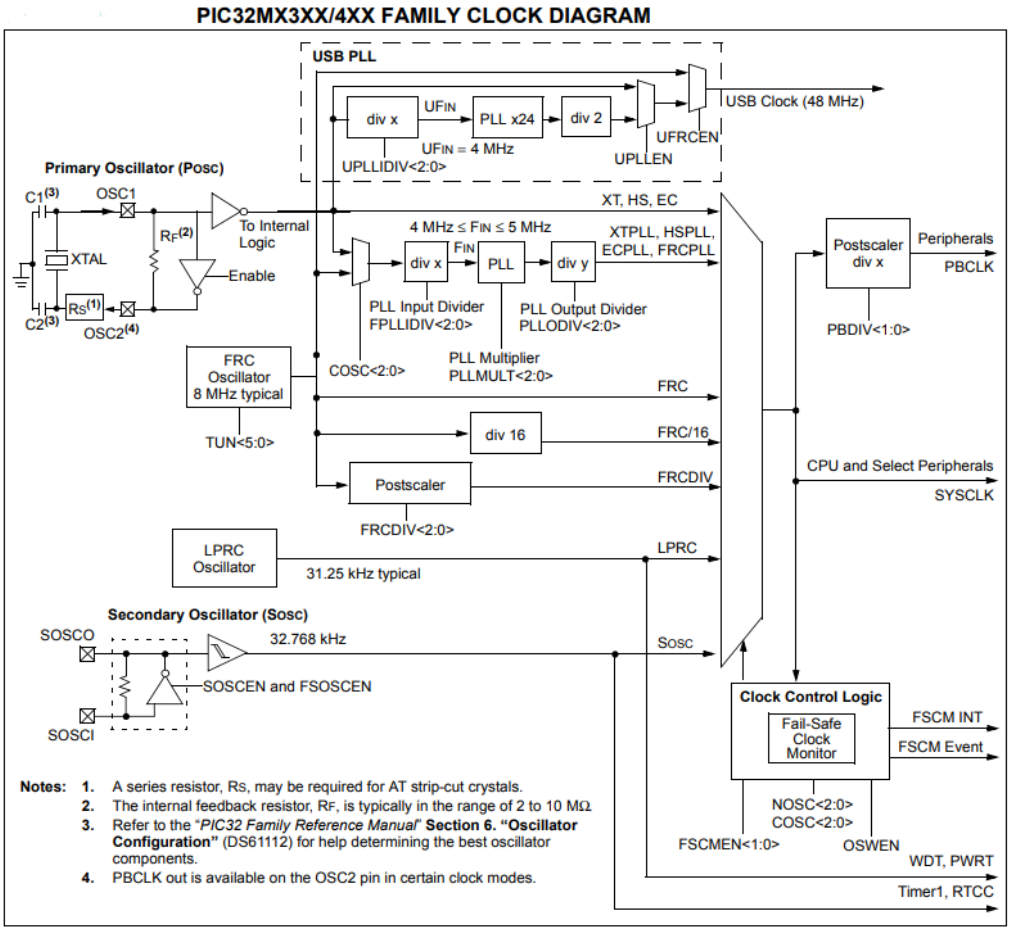


Figura 36: Diagrama de relojes del PIC32MX

Para los objetivos de este proyecto se debe comenzar configurando los registros de control adecuados para obtener las frecuencias del oscilador del sistema (*SYSCLK*), del oscilador de periféricos (*PBCLK*) y del USB (*USB Clock*). Para ello se ha decidido configurar los registros para obtener un reloj del sistema de 60MHz, sin divisores para el oscilador de periféricos -por tanto también 60Mhz- y una frecuencia de 48MHz para el USB. Estos valores seleccionados no son arbitrarios, pues aunque el PIC32MX440 puede configurarse hasta 80MHz y con varios divisores del reloj de periféricos, hemos seleccionado estos valores cuidadosamente, ya que posteriormente tendrán suma importancia a la hora de configurar las UART, dado que se busca generar ratios de bits con el menor error y el generador de ratio de baudios integrado en el chip utiliza estos osciladores para sus cómputos.

```
/** CONFIGURATION ******************************************************/
#if defined(PIC32_USB_STARTER_KIT)
    #pragma config UPLLEN   = ON            // USB PLL Enabled
    #pragma config FPLLMUL  = MUL_15        // PLL Multiplier 60mhz
    #pragma config UPLLIDIV = DIV_2         // USB PLL Input Divider
    #pragma config FPLLIDIV = DIV_2         // PLL Input Divider
    #pragma config FPLLODIV = DIV_1         // PLL Output Divider
    #pragma config FPBDIV   = DIV_1         // Peripheral Clock divisor
    #pragma config FWDTEN   = OFF           // Watchdog Timer
    #pragma config WDTPS    = PS1           // Watchdog Timer Postscale
    #pragma config FCKSM    = CSDCMD        // Clock Switching & Fail Safe Clock Monitor
    #pragma config OSCIOFNC = OFF           // CLKO Enable
    #pragma config POSCMOD  = XT            // Primary Oscillator
    #pragma config IESO     = OFF           // Internal/External Switch-over
    #pragma config FSOSCEN  = OFF           // Secondary Oscillator Enable (KLO was off)
    #pragma config FNOSC    = PRIPLL        // Oscillator Selection
    #pragma config CP       = OFF           // Code Protect
    #pragma config BWP      = OFF           // Boot Flash Write Protect
    #pragma config PWP      = OFF           // Program Flash Write Protect
    #pragma config ICESEL   = ICS_PGx2      // ICE/ICD Comm Channel Select
```

Figura 37: Código de configuración a 60 MHz

En la porción de código de la figura 37 se pueden apreciar los registros de control iniciales, en este caso el multiplicador *PLL* de 15x y el divisor por 2 hacen que -partiendo de los 8 MHz del cuarzo- consigamos un oscilador del sistema establecido en 60 MHz.

Otro de los pasos importantes de la configuración es la definición de pines, para ello se ejecutan algunas funciones de inicialización que configurarán los pines utilizados correspondientes a los puertos del micro, seleccionando los registros apropiados para marcar estos como pines de salida o de entrada o colocarlos a nivel alto o bajo según se requiera, todo ello se realiza mediante los registros *PORTx, TRISx*, y *LATx*.

```
/* Define the LCD port pins */
#define E_PIN               LATBbits.LATB1   //d6
#define RW_PIN              LATBbits.LATB2    //d5
#define RS_PIN              LATBbits.LATB15
#define LCD_DATA_BITS       (0xF0u)
#define LCD_PORT_IN         PORTD
#define LCD_PORT_OUT        LATD


#define E_PIN_DIR           TRISBbits.TRISB1    //d6
#define RW_PIN_DIR          TRISBbits.TRISB2    //d5
#define RS_PIN_DIR          TRISBbits.TRISB15
#define LCD_PORT_DIR        TRISD
```

Figura 38: Porción de código de configuración de pines

- Baud Rate Generator:

El generador integrado del ratio de baudios de UART dispone de un *timer* de 16 bits que se programa mediante el registro *UxBRG*, a su vez el registro *BRGH* establece un divisor por 16x para velocidad estándar o por 4x para alta velocidad. En base a estos registros y considerando que nuestro sistema pretende ser preciso a alta velocidad, se ha seleccionado *BRGH=1* y se han de aplicar las siguientes ecuaciones para calcular el correspondiente valor del registro *UxBRG* acorde a la velocidad requerida:

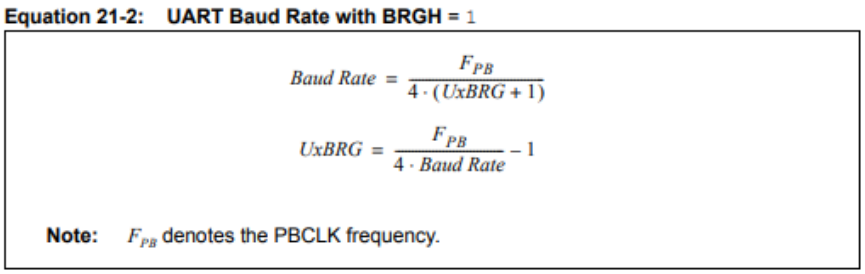


Figura 39: Ecuaciones UxBRG

Una vez realizados los cálculos para establecer las velocidades de transmisión requeridas para el sistema y configurado con 60MHz de F_{PB}, se ha establecido un *array* llamado *baudrates* con doce valores para el registro *UxBRG*, que corresponden con doce valores comunes de configuración y que mantienen un mínimo error, entre un máximo de 1 Mbps y un mínimo de 2400 bps. Estos valores se utilizarán en el código para configurar las UART de acuerdo al parámetro seleccionado por el usuario/a, para reproducir la velocidad de transmisión requerida.

Cabe destacar que esta configuración seleccionada en base a 12 parámetros es totalmente escalable y personalizable a cualquier otro *baudrate* deseado con tan solo añadir o modificar en el *array* el valor correspondiente al nuevo cálculo del registro *UxBRG*, con lo que se dota a la herramienta de una flexibilidad muy alta en cuanto a la velocidad de transmisión.

```
UINT baudrates[12]={14,15,25,32,64,129,259,390,780,1561,3124,6249}; // baud to UxBRG register
char *Baud_cfg[12]={"1Mb","937.5K","576K","460.8K","230.8K","115.2K","57.6Kb","38.4K","19.2K","9600b","4800b","2400b"};
```

Figura 40: Código del array de baudrates

3.7.3. Comunicación con el PC

El PIC32MX dispone de un puerto USB de alta velocidad compatible con USB 2.0 y que utilizaremos para comunicarnos con nuestro dispositivo a través del PC. Para ello se utilizará de base los modelos disponibles en las *Microchip Libraries for Aplications (MLA)* [46] que contiene librerías para diversas funciones, con ejemplos de aplicación sencillos que sirven de ayuda para desarrollar proyectos más complejos.

En este caso se ha hecho uso de las librerías *USB CDC (Comunnication Device Class)* que emulan un puerto serie COM en el PC, es decir, el administrador de dispositivos del PC detectará el microcontrolador como un puerto serie COMx virtual.

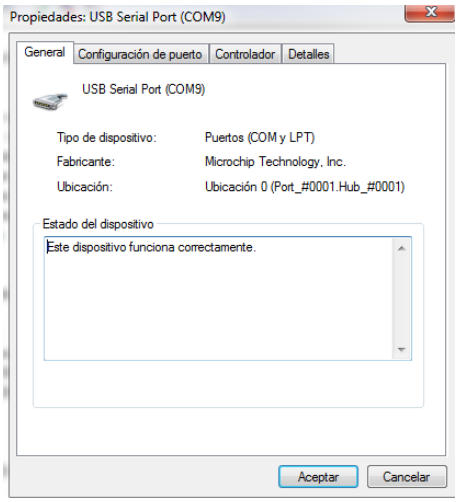


Figura 41: Puerto USB to COM virtual en Windows

En la nota de aplicación *AN1164* [47] publicada por *Microchip* se amplían los detalles para usar las librerías, con la definición de las funciones y las diferentes opciones de personalización que ofrece. En este caso de uso, esperaremos el estado del USB en *CONFIGURED_STATE* y a partir de aquí, se dará por conectado el USB, si llegan datos por el puerto van entrando en el *array USB_Out_Buffer[]* y encolando a su vez en otro *array* que se ha creado, llamado *USB_temp_Buffer[]* que usaremos de buffer temporal para almacenar los mensajes. También cabe destacar que los datos que entran en el buffer del USB se vuelven a enviar al PC a modo de eco, por lo que en el PC se tendrá

feedback de que los datos se han recibido correctamente, y será de utilidad posteriormente -en el software de aplicación PC- para colacionar la comunicación y el control de errores.

A partir de aquí se ha definido la función *SetConfig(),* que irá evaluando los mensajes recibidos en función de su contenido. Para discriminar el contenido de los mensajes recibidos se ha establecido una codificación que hará distinguir qué tipo de contenido se recibe y como tratarlo.

```
/****************Set de la Configuracion en la NVM ******************/
void SetConfig(void){
   if (page>=1 ) {          // Se a recibido al menos una pagina en el buffer USB
          page++;


       if (page > 32) {    // 32 pag de 64bits es el tamaño max, se evalua que se ha llegado y se procesa

           switch (USB_temp_Buffer[0])    // Se evalua el primer bit del buffer usb para identificar que recibimos
           {
               //char caso = USB_temp_Buffer[0];
               case CODE_MSG1:          // se ha recibido el mensase1 a enviar en ASCII
                   NVMErasePage((void *)NVM_PROGRAM_PAGE); // inicializa NVM borrando 4096, es la pag 0
                   hex_decode(USB_temp_Buffer,strlen(USB_temp_Buffer),Uart_Msg_Hex); // paso ascii a hex
                    page=0;
                    size_msg = (count-2)/2; // tamaño del msg hex en Byte menos 2byte de Id de msg y /2 pues char ascii
                    count=0;
                    // Write words starting at Address NVM_PROGRAM_PAGE
                    DelayMs(2);
                    NVMProgram (((void *)NVM_PROGRAM_PAGE), (const void*) Uart_Msg_Hex, size_msg, (void *) 0xA0004000);
                    DelayMs(2);
                   // Verify if data matches
                   if(memcmp(Uart_Msg_Hex, (void *)NVM_PROGRAM_PAGE, size_msg)) //error
```

Figura 42: Porción de la función SetConfig( )

Básicamente se pueden recibir dos tipos de mensajes, uno con los datos a simular, y otro con la configuración del dispositivo. El mensaje de datos contendrá una cadena de caracteres que son los que se enviarán por el puerto RS-232 o RS-422. Este mensaje puede contener hasta 1024 bytes y se identifica porque viene precedido por el carácter ASCII "*Ð*" que corresponde a *0xD0* en hexadecimal. Si se recibe un mensaje sin este carácter precediéndolo o con cualquier otro diferente se entenderá que es un mensaje no válido y generará un error, salvo si se trata de un mensaje de configuración.

Los mensajes de configuración del dispositivo se envían en una palabra de 4 Bytes, como por ejemplo *0xABA1A3E1*, donde se codifica el puerto a usar, la paridad y numero de bits, el *bitrate*, la separación entre mensajes y si se incluirá o no contador de mensajes. El mensaje de configuración enviado deberá comenzar por *0xAA, 0xAB*, *0xBA, 0xBB*, que codificará el tipo de puerto

RS-232 o 422 y si incluirá contador o no. Cualquier mensaje que comience por otra codificación será descartado y generara un error.

La descripción completa de los mensajes de configuración, su posición y su significado se encuentra detallada en el Anexo III y posteriormente se tratará también en otros capítulos de esta memoria, pues al igual que el PIC32 interpreta inequívocamente estos mensajes, el software del PC tendrá que ser capaz de generarlos y enviarlos.

Si el mensaje recibido por el PIC32 comienza por el carácter ASCII "Ð" se entenderá entonces que a continuación viene el mensaje a simular y se procesará adecuadamente. El PC enviará este mensaje codificado en ASCII y en el PIC32 se convierte a hexadecimal para almacenarlo en la memoria no volátil. Para entender este proceso debemos comentar que un carácter de la tabla ASCII ocupa un Byte, cuando en realidad solo representa un solo carácter de una cadena en hexadecimal, por ejemplo: la letra "*A*" mayúscula en ASCII corresponde a *0x41* en hexadecimal por lo que el PC en realidad para enviar *AA* enviará *0x4141*, en el PIC frente a un carácter ASCII que el usuario/a envía por el PC se convertirá a hexadecimal para almacenarlo y ocupará la mitad, por lo que del mensaje *AA* enviado por el usuario/a como *0x4141* nosotros almacenaremos *0xAA* en la memoria no volátil.

Para realizar esta labor se ha programado la función *hex_decode( )* que pasa la cadena recibida en ASCII a Hexadecimal y lo ordena adecuadamente en *Little-endian* para almacenarlo en otro *array* llamado *Uart_msg_hex[ ],* que posteriormente será almacenado en las direcciones de memoria de 32bits que maneja la memoria no volátil del PIC32MX.

```
//******** Convierte ASCII to BIN(HEX) y lo mete en 32 bits /
UINT* hex_decode(const char *in, size_t len, UINT *out)
{
        unsigned int i, t, hn, ln, mn2, mn3, mn4, mn5, mn6, mn7;

        for (t = 0,i = 1; i < len; i+=8,++t) {

                hn = in[i] > '9' ? in[i] - 'A' + 10 : in[i] - '0';
                ln = in[i+1] > '9' ? in[i+1] - 'A' + 10 : in[i+1] - '0';
                mn2 = in[i+2] > '9' ? in[i+2] - 'A' + 10 : in[i+2] - '0';
                mn3 = in[i+3] > '9' ? in[i+3] - 'A' + 10 : in[i+3] - '0';
                mn4 = in[i+4] > '9' ? in[i+4] - 'A' + 10 : in[i+4] - '0';
                mn5 = in[i+5] > '9' ? in[i+5] - 'A' + 10 : in[i+5] - '0';
                mn6 = in[i+6] > '9' ? in[i+6] - 'A' + 10 : in[i+6] - '0';
                mn7 = in[i+7] > '9' ? in[i+7] - 'A' + 10 : in[i+7] - '0';
                out[t] = (mn6<< 28) | (mn7 << 24)| (mn4 << 20)| (mn5 << 16)|
                         (mn2 << 12)| (mn3 << 8) | (hn << 4)| ln;
        }

        return out;
}
```

Figura 43: Código de la función hex_decode()

Con los mensajes de configuración enviados desde el PC ocurrirá algo parecido, pero en este caso se enviará el carácter ASCII y se almacena como tal en hexadecimal sin ninguna conversión, pues simplemente se enviarán caracteres especiales ASCII como por ejemplo "ª" que corresponde a *0xAA* o el carácter ASCII "«" que corresponde a *0xAB* interpretando tales directamente como diferentes opciones de configuración válidas.

3.7.4. Almacenamiento y lectura en memoria no volátil

La sección 5 "*Flash Programming*" del manual de referencia de la familia de microcontroladores PIC32MX describe en profundidad la estructura, registros y las operaciones a realizar para las lecturas y escrituras en memoria no volátil.[48]

La estructura de memoria flash del PIC32MX se compone de filas que forman una página. Una fila contiene 128 palabras de 32 bits o 512 Bytes y un total de 8 filas forman la página, que por tanto tiene 8 x 512 Bytes=4096 Bytes. Este dato es significativo pues para las operaciones de escritura en direcciones ya escritas es indispensable borrar la memoria previamente y el PIC32MX solo permite borrar la flash por páginas, o sea en bloques mínimos de 4096Bytes.

Sin embargo, la escritura de la memoria flash permite la programación por palabras de 32bits (4 bytes) o por filas de 64 o 128 palabras, pero como se ha comentado antes para volver a escribir una dirección previamente escrita se requerirá borrar toda la página. Por otro lado las operaciones de memoria en flash son bloqueantes y hacen que el microcontrolador espere a que se haya escrito la memoria, por tanto el PIC32 no ejecutará ninguna instrucción ni responderá a interrupciones hasta que no haya finalizado el ciclo de programación.

Con estas premisas una vez establecida la configuración o el mensaje para almacenar en la NVM se realiza un borrado de una página de la misma, estableciendo una dirección inicial en el rango de *0x9D000000* que -como se especifica en la figura 7- es la región virtual de memoria donde se ubica la *flash* programable, para ello se utiliza la función *NVMErase( )* y posteriormente se escriben los datos con la función *NVMProgram( ).*

```
case CODE_MSG1:         // se ha recibido el mensase1 a enviar en ASCII
    NVMErasePage((void *)NVM_PROGRAM_PAGE); // inicializa NVM borrando 4096, es la pag 0
    hex_decode(USB_temp_Buffer,strlen(USB_temp_Buffer),Uart_Msg_Hex); // paso ascii a hex
     page=0;
     size_msg = (count-2)/2; // tamaño del msg hex en Byte menos 2byte de Id de msg y /2 pues char ascii son
     count=0;
     // Write words starting at Address NVM_PROGRAM_PAGE
     DelayMs(2);
     NVMProgram (((void *)NVM_PROGRAM_PAGE), (const void*) Uart_Msg_Hex, size_msg, (void *) 0xA0004000);
     DelayMs(2);
    // Verify if data matches
    if(memcmp(Uart_Msg_Hex, (void *)NVM_PROGRAM_PAGE, size_msg)) //error
    {
    // If not turn led1 on to indicate an error
    //    mLED_1_On();
       LCD_SetPosition(LINE_ONE);
       printf("Error writing cfg");
    }
else {
    NVMWriteWord((void*)(NVM_PROGRAM_PAGE + 0x420), CODE_OK); // escribe ok en 3_c420
    NVMWriteWord((void*)(NVM_PROGRAM_PAGE + 0x424), size_msg); // escribe en nvm tamaño de msg
    FlagConf= FlagConf | 0b01; // MSG Config recorded
    LCD_SetPosition(LINE_ONE);
    printf("Message received");
    }
```

Figura 44: Código de escritura de mensaje en NVM

Para el caso del mensaje con los datos a simular se escribirá en la zona seleccionada y tras la escritura se realiza una verificación de esta, si la escritura ha sido correcta se procede a escribir dos detalles más: el tamaño del mensaje escrito, -ya que este puede ser variable y se necesitará después saber cuántos bytes leer- y un mensaje de *CODE_OK,* que indicará si existe un

mensaje válido escrito en NVM cada vez que reiniciemos. Al finalizar las escrituras -si todo ha ido correcto- el display LCD mostrará el texto "*Message received*"

```
case CODE_CONF232:  // Se ha recibido la configuracion de bitrate, paridad etc
case CODE_CONF422:
case CODE_CONF232_C:
case CODE_CONF422_C:
    NVMErasePage((void *)NVM_PROGRAM_PAGE+ (NVM_PAGE_SIZE*2)); // inicializa pag 2 NVM borrando 4096
    size_msg = count/2;
    page=0;
    count=0;
    // Write bytes starting at Row Address NVM_PROGRAM_PAGE
    NVMProgram (((void *)NVM_PROGRAM_PAGE)+ NVM_PAGE_SIZE*2, (const void*) USB_temp_Buffer, 4, (void *) 0xA0004000);
    DelayMs(2);
    // Verify if data matches
    if(memcmp(USB_temp_Buffer, (void *)NVM_PROGRAM_PAGE+NVM_PAGE_SIZE*2, 4))
    {
      // If not turn led1 on to indicate an error
      LCD_SetPosition(LINE_ONE);
      printf("Error writing cfg");
    }
    FlagConf = FlagConf | 0b10;  // Baudrate config recorded
    //LCD_WriteCmd(CLEAR_DISPLAY);
    LCD_SetPosition(LINE_ONE);
    printf("Baudrate cfg rcv");
    break;
```

Figura 45: Código de escritura de configuración en NVM

De igual manera la configuración referente al puerto a usar, *baudrate* y demás parámetros del simulador se escribirá en la siguiente página de memoria, con el objeto de poder borrar y escribir el mensaje o la configuración de manera independiente. Para ello se comenzará borrando la siguiente página y se escribirá esta codificación de configuración que serán siempre 4 Bytes, si la verificación de la escritura en NVM es correcta se mostrará el texto "*Baudrate cfg rcv*" en el LCD.

A partir de aquí ya se encuentra la configuración y los datos a enviar escritos en la memoria no volátil, y en el momento en que se desconecte el USB o se reinicie el dispositivo (USB=0 en el gráfico de estados de la figura 35) se procederá a leer y aplicar la configuración presente en la NVM.

Para esta labor se ha programado la función *ApplyConfig( )* que se encarga de leer las direcciones de memoria no volátil mediante punteros. La función primeramente acude a la dirección de memoria donde se espera encontrar el mensaje *CODE_OK* y si esto existe significa que hay datos y moverá el puntero a la siguiente dirección para obtener el tamaño del mensaje en bytes.

Conociendo el tamaño ya se puede acudir a la dirección adecuada a copiar ese tamaño de datos en el *array Uart_Byte_Hex[ ]* para su posterior proceso.

```
/****************lectura y apply de la Configuracion almacenada en NVM ******************/

void ApplyConfig(void){

    DWORD* fptr = (DWORD*)(NVM_PROGRAM_PAGE + 0x420);
    unsigned short int err;
    dato = *fptr++;
    // When a record is saved to NVM, first byte is written as 0x60606060 to indicate CODE_OK
    // that a valid record was saved.
    if(dato == CODE_OK) {  // Message present in NVM
        dato = *fptr++;  // size de msg en pos 0x424
        memcpy (Uart_byte_Hex,(void *)NVM_PROGRAM_PAGE, dato); // Copia los datos a enviar a Uart_byte_Hex
        fptr = (DWORD*)(NVM_PROGRAM_PAGE+ (NVM_PAGE_SIZE*2)); // salta a pagina 2 donde está el config de baudrates
        baud_conf = *fptr;
        baud_conf_byte = (BYTE)baud_conf;
        if (baud_conf_byte == 0xAA || baud_conf_byte == 0xBA){ // si los LSB es AA hay baud config ok de rs232, si BA
            if (baud_conf_byte == 0xBA){
                msg_num_on=TRUE;  // Contador de mensajes ON
            }
            else
                msg_num_on=FALSE;  // Contador de mensajes OFF
            BYTE conf_array[sizeof(DWORD)];  // Array con campos de baudr, stop etc
            memcpy (conf_array,&baud_conf,sizeof(DWORD));
            UINT i;
            Delay_msg = conf_array[2];
            gap_data(Delay_msg);
            for (i=0xE0;i<0xEc;i++){  // bucle para obtener bit rate NVM entre 0xE0 y 0xEC
                if (conf_array[3]== i){
                    Baudrate = i;
                    Baudrate = Baudrate - 224;// Baudrat es la posicion de array de baudrates[]]
                }
            }
```

Figura 46: Porción de la función ApplyConfig()

A continuación se acude la dirección donde se encuentra la configuración y se decodificarán los 4 bytes que contienen las opciones de configuración, copiándolos a *Conf_array[ ]* y aplicando los valores obtenidos a la configuración de los puertos UART del microcontrolador. Con la función *OpenUARTx( )* se configurarán los correspondientes valores de puerto, *bitrate*, paridad, numero de bits etc. tomando especial importancia el valor *Baudrate* que escribirá el registro *UxBRG* del *baudrate generator* calculado anteriormente, y junto con la función *PrintUxcfg(),* se procederá a mostrar la configuración aplicada en la pantalla LCD.

```
if (conf_array[1]== 0xA4){       // 8bit paridad par 1 stop  8-e-1
     //to do cofig 8 bit even 1 stop
    OpenUART1(UART_EN | UART_BRGH_FOUR | UART_EVEN_PAR_8BIT | UART_1STOPBIT, UART_TX_ENABLE, baudrates[Baudrate]);
    PrintU1cfg();  // Print config applied to LCD
    }
else if (conf_array[1]== 0xB4){  // 8bit paridad par 2 stop
        //to do cofig 8 bit even 2 stop
    OpenUART1(UART_EN | UART_BRGH_FOUR |UART_EVEN_PAR_8BIT | UART_2STOPBITS, UART_TX_ENABLE, baudrates[Baudrate]);
    PrintU1cfg();
    }
else if (conf_array[1]== 0xA5){  // 8bit paridad impar 1 stop
        //to do cofig 8 bit odd 1 stop
    OpenUART1(UART_EN | UART_BRGH_FOUR |UART_ODD_PAR_8BIT | UART_1STOPBIT, UART_TX_ENABLE, baudrates[Baudrate]);
    PrintU1cfg();
    }
else if (conf_array[1]== 0xB5){  // 8bit impar 2 stop
        //to do cofig 8 bit odd 2 stop
    OpenUART1(UART_EN | UART_BRGH_FOUR |UART_ODD_PAR_8BIT | UART_2STOPBITS, UART_TX_ENABLE, baudrates[Baudrate]);
    PrintU1cfg();
    }
```

Figura 47: Configuración de UART en función ApplyConfig()

En este momento el dispositivo se encuentra con la UART correspondiente configurada según los parámetros programados y con los datos a enviar cargados en el *array Uart_byte_Hex[ ],* por lo que se devolverá el control al bucle principal para proceder a la transmisión por RS-232 o RS-422 según se haya seleccionado UART1 o 2.

3.7.5. Generación de mensajes en puertos UART

Una vez configurado el sistema se llama a la función *Send_message( ).* Esta función lo primero que evalúa es si se debe incluir un contador de mensajes previo a cada mensaje enviado. Si esta funcionalidad ha sido configurada se incluirá un contador incremental de 8 bits al comienzo de cada mensaje que facilitará identificar si hay pérdida de mensajes u otros comportamientos anómalos.

```
void Send_message()  // Envia el mensaje a al puerto configurado con el gap y contador segun config
{
    int i;
        if (msg_num_on){  // Si esta activo se añade contador de mensajes de 8bit previo a cada msg
            DelayUs(1);

            if (!port_422)
                WriteUART1(msg_num++);
            else
                WriteUART2(msg_num++);
            }
            if (!port_422){
        for (i=0;i<dato;i++){

            while (!UARTTransmitterIsReady(UART1));
              WriteUART1(Uart_byte_Hex[i]);
            }
               while(BusyUART1());
                   delay_us(gap_delay);
            }
            else
                {
            for (i=0;i<dato;i++){
            while (!UARTTransmitterIsReady(UART2));
              WriteUART2(Uart_byte_Hex[i]);
            }
            while(BusyUART2());
                delay_us(gap_delay);
            }
```

Figura 48: Función Send_message()

Después la función procede a evaluar el puerto UART configurado y a encadenar en bucle el envío del número de bytes correspondientes por el puerto seleccionado. Se procede a mostrar el mensaje "*Transmiting..."* en el LCD y a comprobar los registros pertinentes de la UART del PIC32MX, pues éste no podrá enviar datos si no está listo o si su cola está llena, para ello se utilizan las funciones *UARTTransmitterIsReady()* y *BusyUART()* para ir concatenando los mensajes que se enviarán con la función *WriteUARTx(),* según la disponibilidad de estos recursos.

Una vez transmitido el mensaje se procede a dejar una separación antes del siguiente mensaje. Esta separación es definida por la configuración de usuario/a y se aplica al final de cada mensaje transmitido mediante la función *delay_us(gap_delay).* Esta función de *timer* es un contador basado en el *Core Timer* del PIC32MX y que se ha incluido en el código del Anexo IV -junto con otras funciones de utilidad- en el archivo fuente llamado *Util.c*. Este temporizador se pone a cero al llamar a la función y cuenta los ciclos del microcontrolador en base al parámetro *gap_delay*, este parámetro toma el valor en microsegundos según la configuración enviada por el usuario/a.

```
void delay_us(unsigned int us)
{
us *= (SYS_FREQ / 2000000 / 1); // Core timer updates every 2 ticks
_CP0_SET_COUNT(0); // Set Core Timer count to 0
while (us > _CP0_GET_COUNT()); // Wait until Core Timer count reaches the number we calculated earlier

}
```

Figura 49: Función delay_us()

Una vez aplicado el retardo el sistema continua transmitiendo en bucle con esta función *Send_message()* hasta que se interrumpa por una nueva condición de configuración.

3.7.6. Control de errores

A lo largo del código programado se han establecido condiciones de error que pueden darse y que son tratadas en la mayoría de las ocasiones mostrando el error con un texto en el display LCD. Las condiciones de error que se pueden mostrar son las siguientes:

- "*No Cfg Available*": Este mensaje se muestra cuando el dispositivo no tiene ninguna configuración en su memoria no volátil, ocurre al iniciar y con USB=0, por ejemplo nada más ser programado el PIC, pues su memoria no volátil está en blanco. También si al iniciar se mantiene el switch pulsado. Se resolverá conectando al PC y configurándolo adecuadamente.

- "*Unknow config!!":* Este mensaje se muestra con USB=1, cuando estando conectado al PC se envían mensajes de configuración que no están definidos en la codificación. Cualquier mensaje enviado por el USB que no comience reflejando la codificación del Anexo III generará este error.

- "*Error writing cfg!!":* Este error ocurre configurando con USB=1, cuando al verificar la escritura en la NVM los valores leídos no coinciden con los escritos. Este error puede deberse a intentar escribir mensajes de tamaño no soportado (no múltiplos de 32bits) o por fallo en el hardware.

- "*Invalid Config!!":* Este error sucede con USB=0, habiendo datos de configuración en la NVM pero alguno puede no estar soportado. Por ejemplo se ha enviado un *baudrate* no soportado. Se solucionará configurando el sistema en base a los parámetros y especificaciones del Anexo III.

3.7.7. Otras funcionalidades

La placa de desarrollo dispone de un pulsador de *reset* y otro de libre uso conectado a un pin digital. Se ha implementado una rutina al comiendo del código que detecta si el pulsador se mantiene apretado al iniciar. Si esta situación ocurre se mostrará el mensaje “DELETING CFG...” en el display LCD y se procederá a borrar toda la configuración de la memoria no volátil, pasando el dispositivo a la configuración inicial.

3.7.8. Programación del PIC y pruebas preliminares

Una vez compilado el firmware que correrá en el microcontrolador se procede a programarlo con el entorno MPLAB X y el programador MPLAB SNAP. Con las funciones de depuración que ofrece este programador es posible comprobar que los comportamientos son los esperados, estableciendo puntos de ruptura y comprobando valores de variables y demás.

Figura 50: Hardware tras la primera programación

Lo primero que se aprecia después de programar el PIC32 es que no hay configuración alguna y en el LCD se muestra el error "*No Cfg Available*", todo según lo previsto pues la NVM está vacía.

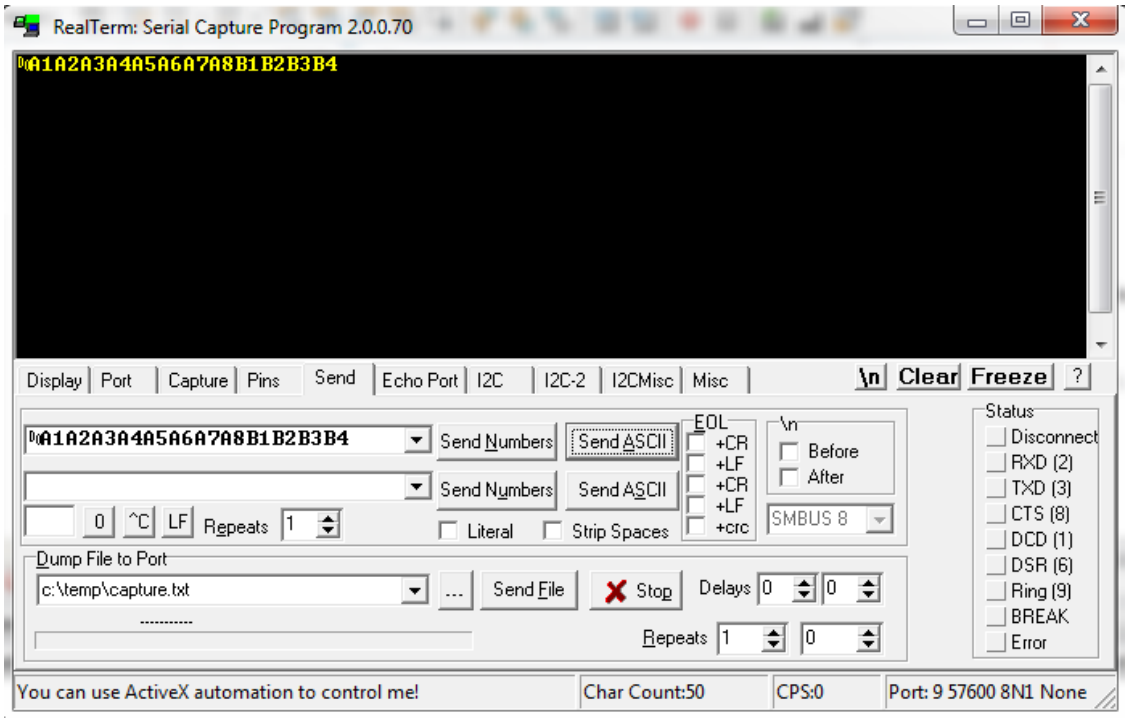


Figura 51: Pantalla del programa Realterm

Se ha utilizado un software libre de terminal -como *Realterm*- para enviar comandos por el puerto COM virtual que nos genera el PIC32 al conectarlo al USB del PC. Usando puntos de ruptura se comprueba con el depurador que los mensajes se reciben según lo esperado, se examina el *array Uart_msg_hex[]* y se observa que los valores se corresponden con lo configurado y con lo escrito en la NVM (Fig.52).

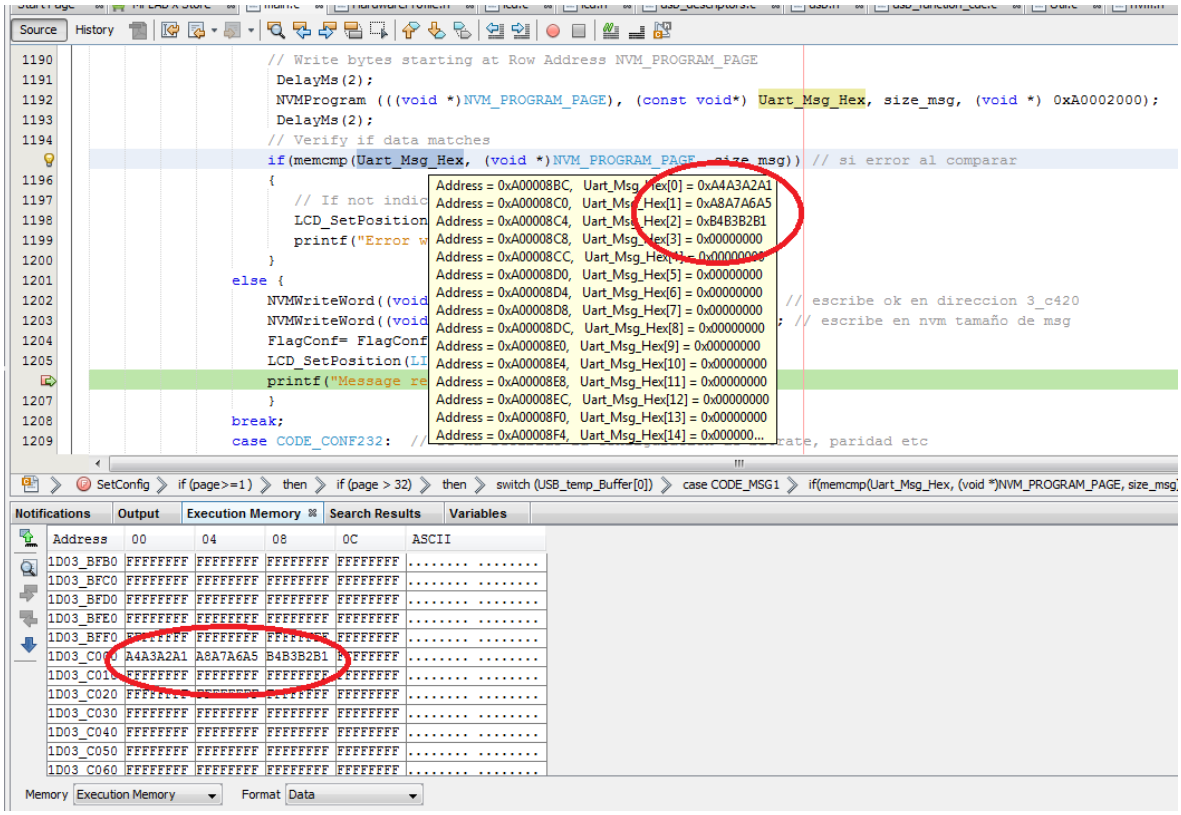


Figura 52: Pantalla del IDE MPLAB X en depuración

De igual manera, generando el resto de mensajes de configuración se comprueba que los comportamientos al enviar los mensajes con *Realterm* son los esperados, se verifica que se *parsean* los mensajes de configuración, se escriben en memoria y se aplican los cambios en los puertos.

3.8. Diseño del software de configuración de PC

El entorno de desarrollo y lenguaje de programación *Processing* es extremadamente intuitivo y permite la realización de aplicaciones gráficas de manera sencilla. Sin dedicar un sobreesfuerzo a la programación del entorno gráfico de usuario/a permite al programador/a concentrarse en las funciones a realizar de manera eficiente. A continuación se describe el proyecto realizado con *Processing* para realizar la aplicación de configuración.

3.8.1. Diagrama de flujo de la aplicación

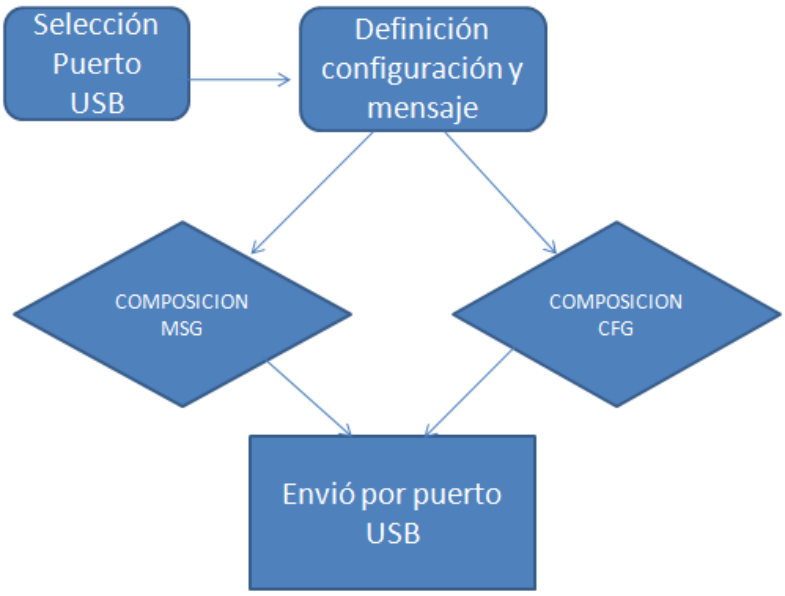


Figura 53: Diagrama de flujo de la aplicación de PC

En la figura 53 se representan los diferentes bloques que recorrerá la aplicación, cuyo código fuente puede encontrarse completo en el Anexo IV. Este flujo comienza en un módulo que conectará la aplicación de PC con el microcontrolador a través del puerto USB, una vez abierto el puerto se deberán establecer las configuraciones requeridas y los datos que contendrá el mensaje a simular por el PIC. Con estos datos se compondrá un mensaje de configuración y de datos que será enviado al PIC32MX para comenzar a simular, una vez desconectado del PC.

3.8.2. Comunicación con el microcontrolador

Processing permite comunicarse con puertos serie importando la biblioteca *processing.serial.** que se encuentra integrada en el propio IDE. Esta biblioteca lee y escribe datos hacia y desde dispositivos externos, de byte en byte y permite que los dos dispositivos envíen y reciban datos. En este caso se buscará comunicar con el puerto COM virtual que crea el PIC32MX al conectarlo por USB al PC.

El método *serial.list()* devolverá el listado de los puertos disponibles que seleccionaremos mediante botones y con el botón "*Connect"* se procede a abrir el puerto COMx que corresponda al PIC32 para proceder a enviar las configuraciones.

```
// create the buttons
btn_serial_up = new Buttons("^", 825, 505, 40, 20);
btn_serial_dn = new Buttons("v", 825, 535, 40, 20);
btn_serial_connect = new Buttons("Connect", 880, 480, 100, 25);
btn_serial_disconnect = new Buttons("Disconnect", 880, 510, 100, 25);
btn_serial_list_refresh = new Buttons("Refresh", 880, 540, 100, 25);

// get the list of serial ports on the computer
serial_list = Serial.list()[serial_list_index];
```

Figura 54: Código de botones y listado de puertos

3.8.3. Parámetros de configuración

Processing -al igual que lo haríamos con Java- también permite incorporar librerías de terceros para ampliar las funcionalidades, en este caso se han importado las librerías *G4P* [50] y *ControlP5* [51], ambas de libre distribución bajo licencia GNU GPL. Para incorporarlas se deben descargar y copiar en la carpeta adecuada y nos proveerán de amplias funcionalidades. Con ellas se han creado los desplegables, cuadros de texto, botones y casillas de verificación necesarias para disponer de todas las opciones de configuración.

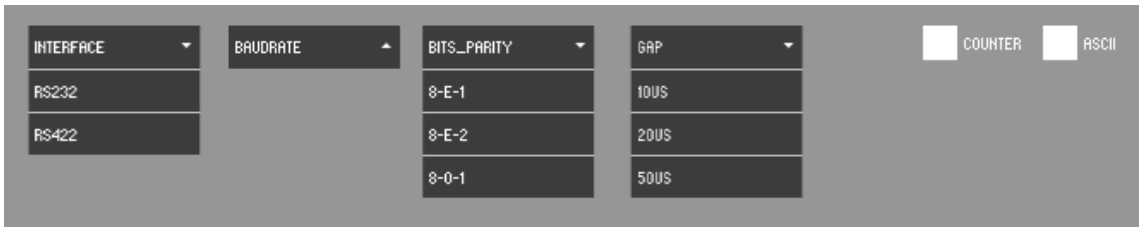


Figura 55: Detalle de los desplegables de opciones

En los desplegables podemos encontrar las selecciones del puerto a usar, el *baudrate*, la paridad, número de bits y el *gap* entre mensajes. También encontraremos dos casillas de verificación, que marcaremos para incluir el contador de mensajes o para enviar el mensaje codificado en ASCII o en hexadecimal. Estos desplegables de selección se evalúan en el programa con funciones *switch-case* y según lo seleccionado codifican la porción adecuada del comando a enviar hacia el PIC.

```
void INTERFACE(int n) {
    print("Interface select :  ");
    println(n, cp5.get(ScrollableList.class, "INTERFACE").getItem(n).get("name"));
   switch(n) {
      case 0:
        Port_cfg= "ª"; //0xAA RS232
        break;
        case 1:
        Port_cfg= "«";  //0xAB RS422
        break;
   }
}

void BITS_PARITY(int n) {
    print("Bits select :  ");
    println(n, cp5.get(ScrollableList.class, "BITS_PARITY").getItem(n).get("name"));
   switch(n) {
      case 0:
        Bits_cfg= "¤";   //0xA4 8-E-1
        break;
      case 1:
        Bits_cfg= "´";  //0xB4 8-E-2
        break;
      case 2:
        Bits_cfg= "¥";  //0xA5 8-O-1
        break;
```

Figura 56: Código de codificación de configuración

En el cuadro de texto se introducirá el mensaje a transmitir, con la limitación de que deberá ser en múltiplos de 4 Bytes, pues como hemos visto anteriormente lo escribiremos en la NVM del PIC32 y esta es su mínima unidad de escritura en memoria.

Con todas estas selecciones se compondrá un comando acorde al Anexo III, que a modo de ejemplo enviaría por el puerto un comando como *0xAAA4A7E1* que significaría configurar:

- *AA* → Puerto RS232 sin contador de mensajes
- *A4* → 8 bits-paridad par-1 bit stop
- *A7* → 1ms de gap entre mensajes
- *E1* → *Baudrate* de 937500bps

En el caso del mensaje de datos en sí, una vez introducido el texto deseado se enviará precedido del identificador *0xD0*. Si se selecciona en hexadecimal los caracteres entre *00* y *FF* se enviarán a continuación codificados tal cual y en mayúsculas, y si se hubiera seleccionado la marca de codificación ASCII el código de la letra o número será convertido al valor hexadecimal correspondiente de la tabla ASCII. Esto se realiza mediante la función

ASCIItoHEX( ), de forma que al ser recibido en el microcontrolador este enviará al puerto serie los caracteres en la codificación seleccionada. Mediante *serial_port.write( ) se envía la codificación desde Processing.*

```
// Send the text to the serial port

public void sendText(String text) {   // Envia el mensaje ascci por el puerto precedido por el identificador
  int resto;
  if (ascii_msg==true) {
  text=(ASCIItoHEX(text));
  }
  else{
  text = text.toUpperCase();
  }
  println(text);
  ll= text.length();
  resto = ll % 4 ;
  if (resto !=0){
    status=2;  // error multiplo 4
    println("Msg debe ser multiplo de 4");
    println(resto);
  }
  else if (serial_port != null) {
    serial_port.write(CODE_MSG1+text);
```

Figura 57: Porción de la función sendText()

A modo de ejemplo el mensaje "*Hola*" en ASCII, se enviaría como *0xD0486F6C61* y para enviar un mensaje directo en hexadecimal este deberá solo contener caracteres de *0 a F* y no marcar la casilla ASCII para conservar la codificación.

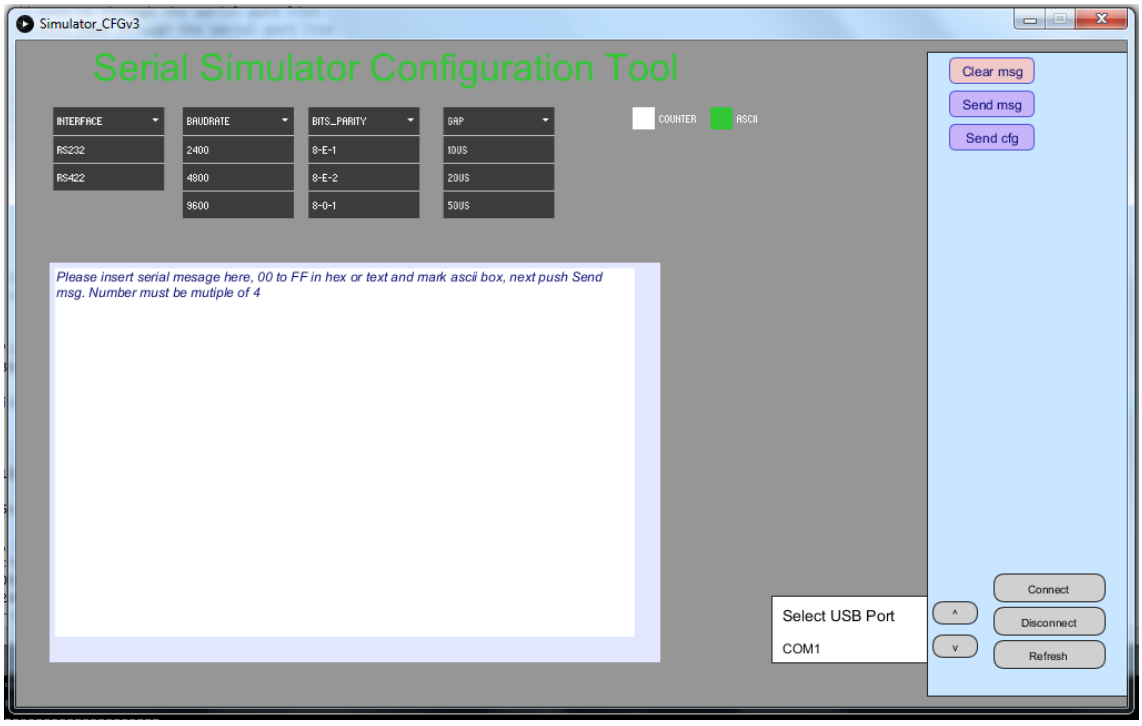


Figura 58: Interfaz de usuario/a del programa

La interfaz de usuario/a tiene el aspecto mostrado en la figura 58, donde se pueden apreciar los desplegables para la selección de configuración, casillas de verificación, botones y el cuadro de texto donde se introduce el mensaje a simular.

3.8.4. Control de errores

En el programa se ha dispuesto de varios mecanismos de control que hacen frente a las excepciones que puedan ocurrir. Estas excepciones se representan en pantalla mediante mensajes de error (figura 59) y se controlan en el código con una estructura *switch-case* mediante la variable *status.*

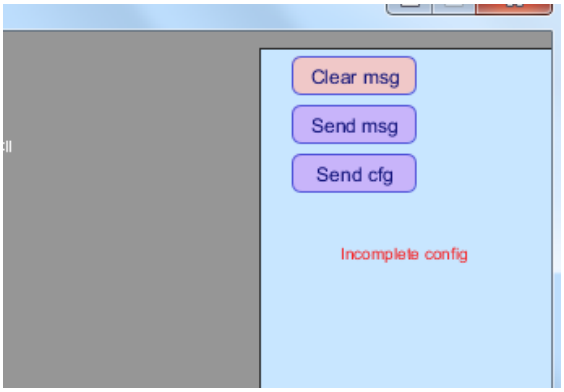


Figura 59: Detalle de error incomplete config

Si el puerto serie no se ha conectado y se pulsa enviar aparecerá el mensaje "*COM port closed*". Igualmente, si se deja alguno de los valores de los desplegables sin completar se muestra el mensaje "*Incomplete config*".

```
switch (status) {
  case 0:
   break;
  case 1:
   text("Incomplete config", width/1.1, 140);
   break;
  case 2:
   text("Message must be mutiple of 4", width/1.1, 140);
   break;
  case 3:
   text("COM port closed", width/1.1, 140);
   break;
  case 4:
   text("Command send", width/1.1, 140);
   break;
  case 5:
   text("Verify COM port ", width/1.1, 140);
   break;
}
```

Figura 60: Rutina de control de errores

En caso de seleccionar un puerto equivocado y no el del PIC32, al tratar de enviar datos se muestra el mensaje "*Verify COM port*" y si el mensaje introducido no es múltiplo de 4 se muestra el mensaje "*Message must be multiple of 4*".

En esta parte del código (Figura 61), también se maneja la respuesta del PIC32 ante un mensaje enviado correctamente desde el PC, pues el PIC devuelve el eco de lo enviado por el puerto USB, y dado el caso se muestra el mensaje *"Command send"* para verificar que el mensaje ha sido correctamente recibido.

```
else if (serial_port != null) {
  serial_port.write(CODE_MSG1+text); // envia mensaje
    delay(100);
    String inBuffer;
   if (serial_port.available()>0){
     inBuffer= serial_port.readString(); // recibe eco de PIC
       if (inBuffer.equals(CODE_MSG1+text)==true){ // si es igual send a rcv es OK
        //println(inBuffer);
         status=4;} // estatus command send
   }
    else status=5;
```

Figura 61: Porción rutina de feedback

Una vez comprobadas todas las funcionalidades del programa y su correcto funcionamiento se ha generado el ejecutable *.exe* desde el menu *Archivo>Exportar Aplicacion...* del IDE de *Processing* y con esto ya se dispone de la aplicación ejecutable y plenamente funcional.

4. Verificación del sistema a cadena completa

4.1. Pruebas funcionales

Se ha establecido una batería de pruebas funcionales, consistente en la utilización de la aplicación de PC junto al simulador, y proceder a comprobar que todas las combinaciones posibles de configuración son recibidas satisfactoriamente, aplicadas y mostradas en el display LCD del hardware y generadas en los puertos correspondientes.

Para verificar que los datos se envían correctamente –una vez simulando las señales- se conectará a un PC por otro puerto serie y se verificará que los mensajes enviados se reciben correctamente en todos el rango de configuraciones posibles, tanto por el puerto RS-232 como por el RS-422. Para ello se dispone de dos conversores de USB que servirán para comprobar la codificación correcta del mensaje y la correcta recepción.

Para completar las pruebas se realizarán las medidas de separación entre mensajes con el osciloscopio, comprobando que efectivamente los retardos de los mensajes se ajustan a los tiempos especificados con precisión. Igualmente se verificará que los niveles de tensión generados por el simulador se ajustan a los estándares simulados.

Dado que el osciloscopio que se ha tenido disponible puede decodificar el protocolo, se comprobará adicionalmente que los mensajes enviados contienen el mensaje adecuado y que las diferentes codificaciones se realizan correctamente.

4.2. Resultados

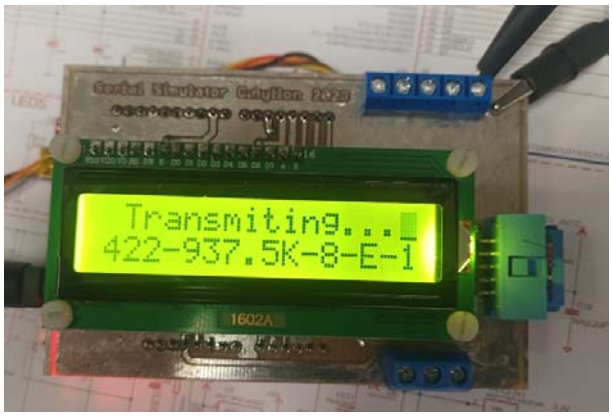


Figura 62: Simulador configurado y transmitiendo RS-422

Tras realizar todas las pruebas se ha comprobado que tanto la parte de aplicación software de PC como el hardware funcionan correctamente a cadena completa. Se ha verificado que las excepciones y errores se procesan con el comportamiento adecuado tanto en el software como en el PIC, mostrándose en la interfaz de la aplicación y en LCD.

Igualmente se ha verificado que las diferentes configuraciones se simulan fielmente, se muestran en el LCD y en otro PC se reciben los mensajes enviados de forma satisfactoria tanto en RS-232 como RS-422 (Fig. 61).

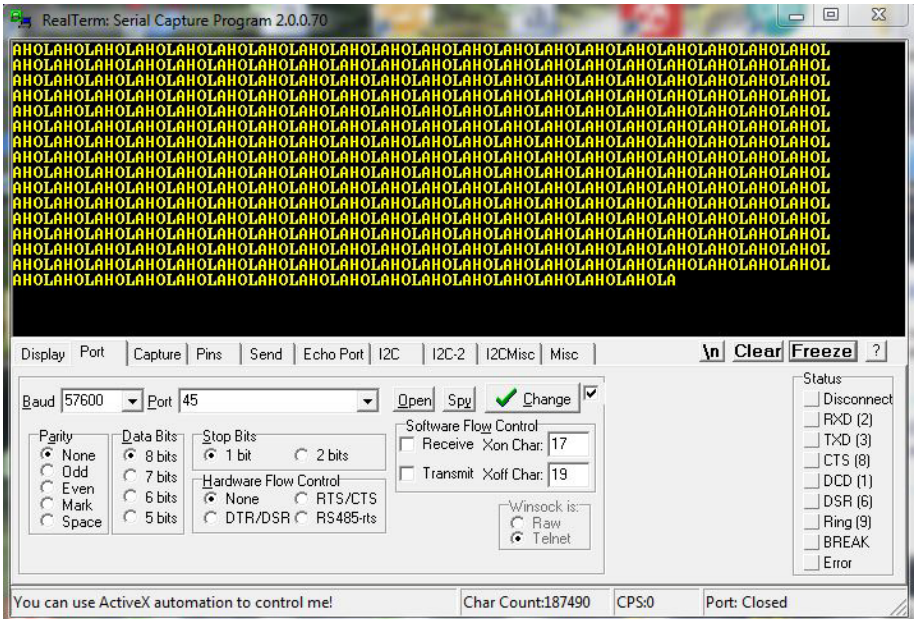


Figura 63: Recepción de mensajes simulados en PC

De la misma forma, en las verificaciones que se han realizado midiendo las señales con el osciloscopio se aprecia que los *gap* de separación entre mensajes cumplen con precisión con las especificaciones y los retardos programados. (Fig.62)

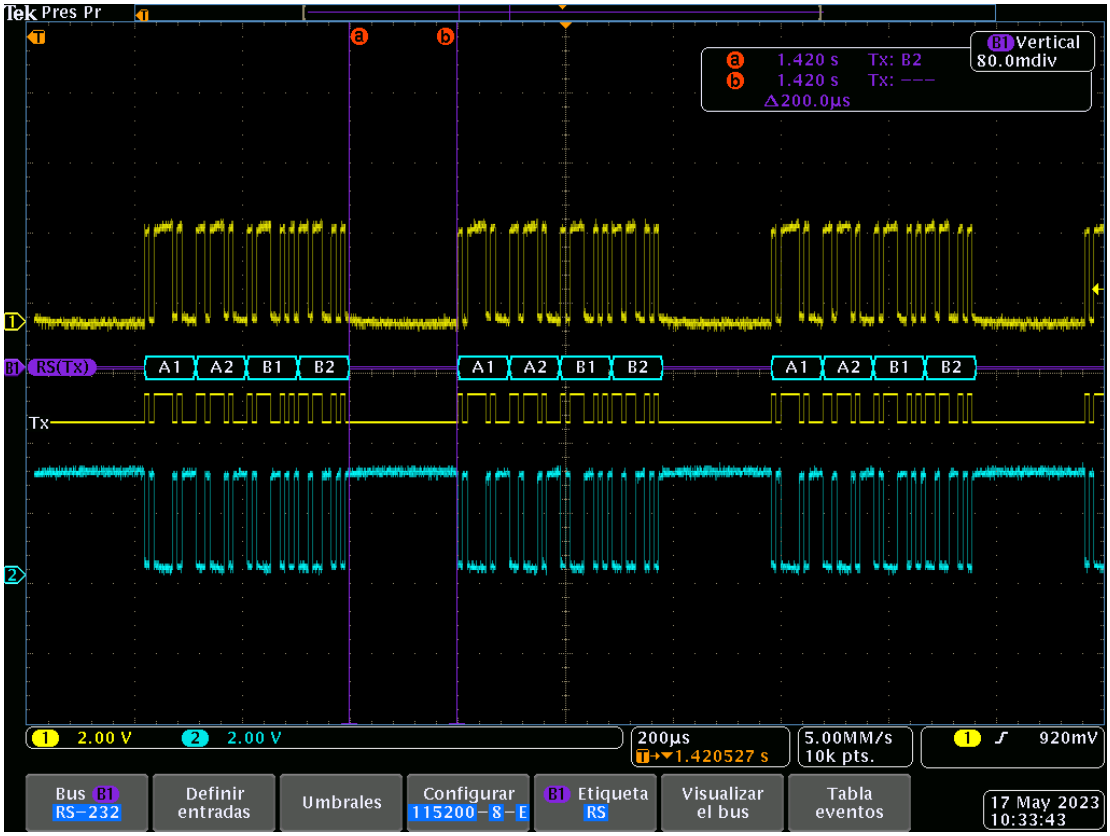


Figura 64: Señal RS-422 a 115200bps-8E1 con 200us de gap

Otro punto verificado es que los niveles de tensión para ambos buses se encuentran dentro de los valores mínimos especificados y que incluso a altas velocidades se envían los datos correctamente manteniéndose los tiempos marcados.(Fig. 64 y 65)

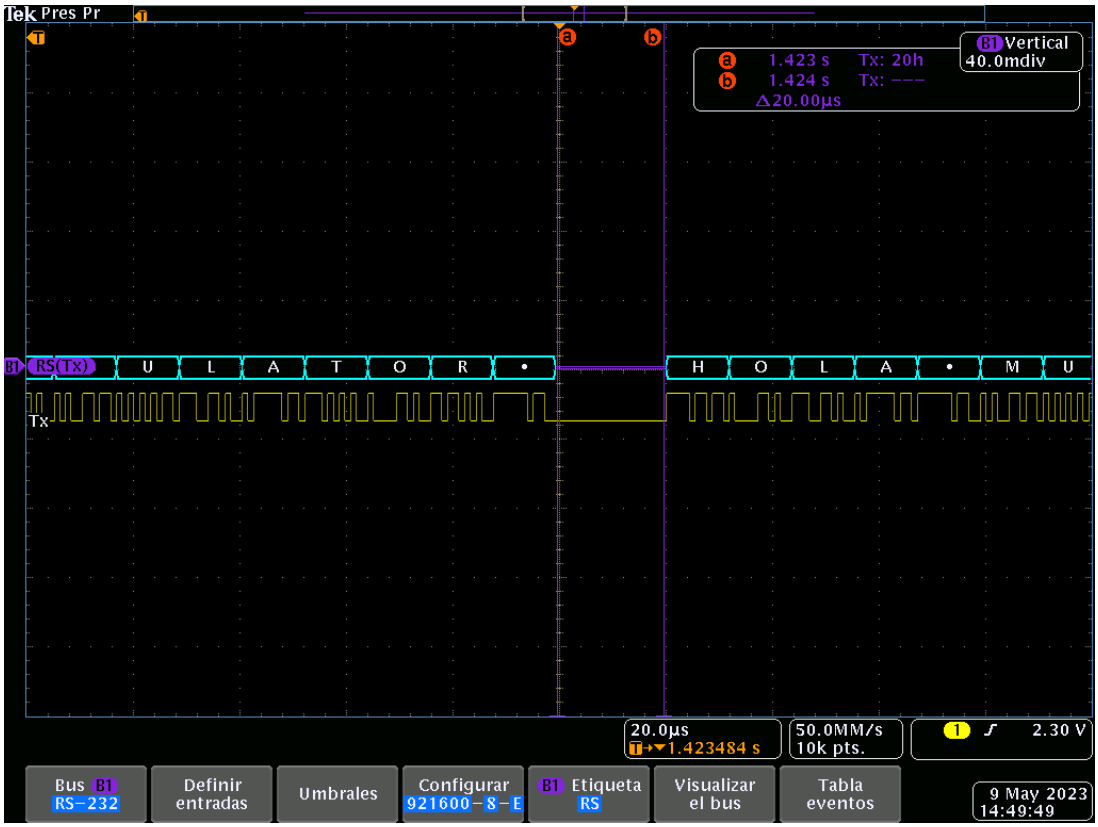


Figura 65: Señal RS232 a 921600bps-8E1 y 20us de separación

También se ha verificado tanto que se admite la máxima carga de pago a 1024 Bytes como que el contador de mensajes de 8 bits funciona correctamente y se numeran los mensajes sin perdidas.(Fig.66)

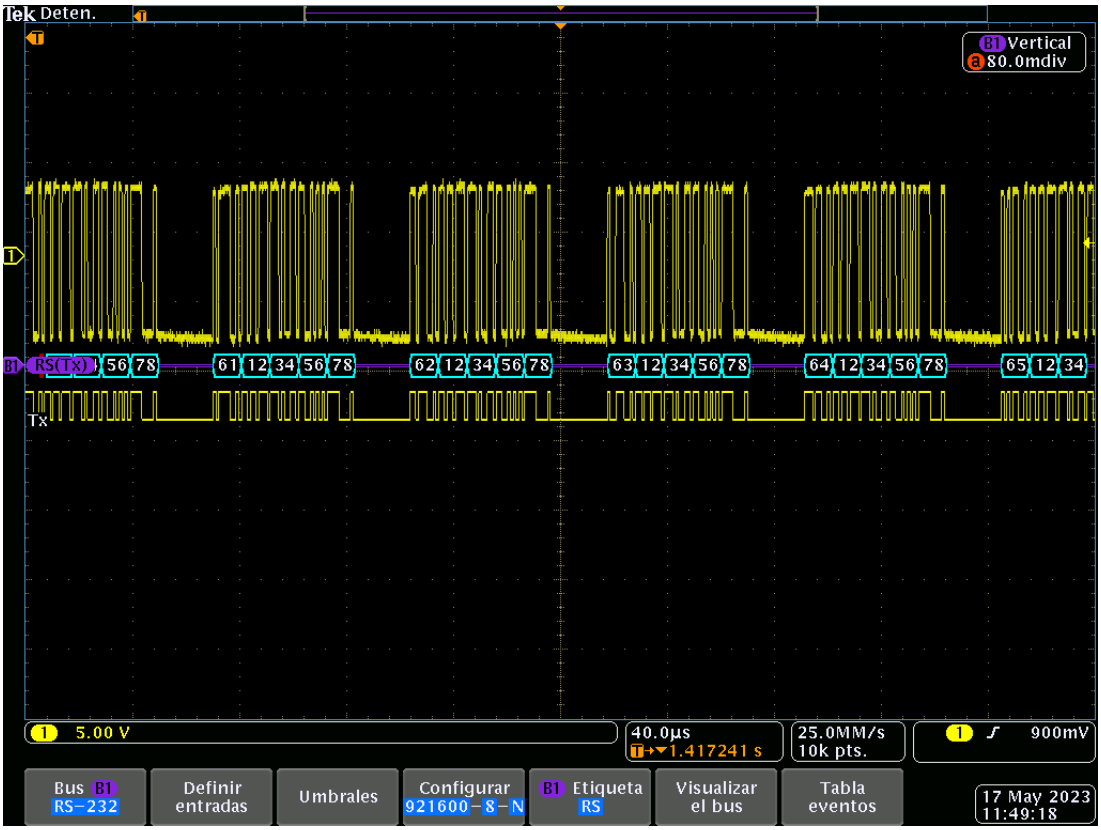


Figura 66: Contador al inicio de los mensajes

4.3. Problemas y soluciones

Durante el desarrollo del proyecto se han ido abordando algunos problemas que se han ido solventando satisfactoriamente y que se detallan a continuación:

- Inicialmente se planteó conectar el display LCD aprovechando el puerto *Parallel Master Port (PMP)* del microcontrolador PIC32, ya que este puerto ofrece librerías y funcionalidades que facilitarían la posterior integración, pero se comprobó que estos pines no están conectados a la placa de desarrollo, por lo que se solucionó moviendo el display a otros pines digitales disponibles y reprogramando el código, estableciendo las rutinas adecuadas para manejar el display LCD en esos pines.
- De igual manera se encontró que la UART2 no estaba conectorizada a ningún conector del *shield* estándar, sino al conector UEXT lateral de la placa, por lo que se solucionó usando el cable y conector IDC empleado en el diseño final.

- Durante las verificaciones previas a la fabricación de la PCB se encontraron algunos errores de diseño que impedían soldar algunos componentes, por lo que hubo que mover algunas pistas a otra capa.
- Se detectaron discrepancias al verificar la escritura en la NVM, dado que el microcontrolador escribe y lee en memoria en formato *Little-endian* y los datos del PC por USB se envían en *Big-endian*. Se solucionó dentro de la función *hex_decode()* que al convertir también ordena los bits por palabras de 32 bits al formato adecuado.
- De igual forma la NVM del PIC32 no se puede escribir en bloques menores de 32 bits, por lo que se establecieron los mecanismos de control para no enviar datos que no sean múltiplos y controlar los errores.
- Al medir los tiempos reales de *gap* entre mensajes se ha comprobado que hay aproximadamente 2us de retardo extra en los tiempos debido al propio tiempo de proceso del bucle y del control de registros UART del microcontrolador, este tiempo se ha medido como constante y se ha establecido como solución un *offset* corrector en los retardos entre mensajes. Esta corrección se aplica hasta los 1000us de *gap*, pues a tiempos mayores se ha estimado despreciable.

5. Conclusiones y trabajos futuros

5.1. Evaluación de los objetivos

A lo largo del desarrollo de este trabajo se han ido evaluando parcialmente los hitos propuestos y los trabajos pendientes por afrontar, durante este trayecto sin duda el desarrollo de toda la arquitectura detallada en el capítulo 3 ha sido la parte más intensa y laboriosa, ya que durante aproximadamente 40 días se han empleado todas las capacidades disponibles, requiriendo estudiar abundante documentación y todo ello tratando de mitigar los riesgos evaluados inicialmente para el proyecto.

Teniendo en cuenta todas las pruebas realizadas y los requerimientos establecidos al inicio del proyecto se puede decir que se han cumplido ampliamente todos los objetivos, ya que se ha conseguido un producto final funcional, sin fallos, que cumple lo especificado y que -si bien es un prototipo- fácilmente podría ser un producto final plenamente funcional. Esta consecución de objetivos es también fiel muestra de la adopción y aplicación de los conocimientos adquiridos durante el Grado y a lo largo de este trabajo final.

Cabe destacar que el coste económico total de los materiales del proyecto ha sido de aproximadamente 130 Euros, lo que sin duda es un coste algo inferior a lo presupuestado, todo ello teniendo en cuenta que se han adquirido algunas herramientas como el programador, que se podrán utilizar durante mucho tiempo en otros desarrollos y pasarían a englobar más una inversión que un gasto.

No han surgido desviaciones importantes de la planificación ni ha habido que aplicar contingencias, pero es satisfactorio comprobar que con el presupuesto sobrante se podría haber absorbido la eventualidad de haber tenido que adquirir algún repuesto.

La metodología seguida ha demostrado ser la adecuada y el desarrollo conjunto de campos tan diferentes como el hardware y software a la vez habiendo solventado todos problemas, retos y desafíos planteados ha sido de especial satisfacción para el autor.

A lo largo del trabajo ha sido necesario introducir algunos cambios menores para garantizar el éxito del trabajo y a veces probar y reescribir porciones de código varias veces para optimizar o conseguir los resultados esperados.

Como conclusión, con este proyecto se ha conseguido disponer de un simulador de comunicaciones Rs-232 y RS-422 a un coste muy bajo, con el que se podrán testar multitud de sistemas y dispositivos con fiabilidad y garantías. Además ha demostrado una facilidad de personalización muy alto para poder introducir pequeños cambios para otras aplicaciones más concretas.

5.2. Líneas futuras

Desde la concepción inicial de este proyecto se han establecido límites en la especificación, en el presupuesto y sobre todo en la planificación con tiempo limitado, todo ello con el objetivo claro de llegar al fin con un producto que cumpla los objetivos descritos, pero sin duda hay otras líneas futuras que se podrían explorar como ampliación o mejora.

Durante el desarrollo del proyecto se han tomado alguna acciones de previsión de escalado, que pueden ser líneas futuras de trabajo, como por ejemplo el utilizar la electrónica de conversión a RS-422 para simular también señales de RS-485 y que requerirían de la ampliación del firmware del PIC y el software de PC, para ello se han dejado conectados los pines de control del driver al microcontrolador. Este desarrollo sin duda ofrecería desafíos adicionales a afrontar, pues el RS-485 permite más de un *master* en la línea y habría que gestionar esta recurrencia en el hardware.

Las conexiones de recepción de los puertos UART no se utilizan, pero se han dejado conectadas, ya que en el futuro se podría utilizar el mismo hardware para alguna utilidad que requiera la recepción de datos, o análisis de ellos y no solo la transmisión.

Otra línea a desarrollar sería la adaptación de una batería recargable para hacer el dispositivo totalmente autónomo en entornos donde no haya alimentación, junto con el diseño de una caja que permita albergar el dispositivo.

6. Glosario

BPS: Bit por segundo.

LCD: Liquid Crystal Display, pantalla de cristal líquido.

PCB: Printed Circuit Board, circuito impreso.

HW: Hardware.

SW: Software.

UART: Universal Asynchronous Receiver Transmitter

IDE: Integrated Development Enviroment, entorno de desarrollo integrado.

PIC: Familia de microcontroladores de Microchip Technologies.

Baudrate: Ratio de bits.

Datasheet: Hoja de datos o características.

WBS: Work Breakdown Structure, diagrama de flujo de tareas.

TTL: Transistor-Transistor Logic, lógica que suele funcionar con 0 a 5V.

FIFO: Fist In First Out, primero en entrar primero en salir.

IC: Circuito integrado o chip.

Buffer: Espacio donde se almacenan datos de manera temporal.

E/S: Entrada-Salida.

FPGA: Field Programable Gate Array, dispositivo lógico programable.

CAD/CAM: Diseño y manufactura asistidos por ordenador.

MIPS: Familia de procesadores con arquitectura RISC.

EEPROM: Memoria de solo lectura programable y borrable eléctricamente.

GPIO: General Purpouse Input/Output.

SMD: Surface Mount Device.

RX: Recepción.

TX: Transmisión.

GND: Ground.

NVM: Non-Volatile Memory.

CFG: Configuración.

MSG: Mensaje.

PLL: Phase-Locked Loop.

7. Bibliografía

[1] *[Especificación]* Autoria: *ANSI/TIA/EIA.* Título*:* "*EIA/TIA-232-F Interface Between Data Terminal Equipment and Data Circuit– Termination Equipment Employing Serial Binary Data Interchange*". *[Publicado: 30/09/1997].*

[2] *[Especificación]* Autoria: *ANSI/TIA/EIA.* Título*:* "*EIA/TIA-422-B Electrical Characteristics of Balanced Voltage Digital Interface Circuits*". *[Publicado: 13/04/1994].*

[3] *[Publicación]* Autoria: *Texas Instruments.* Título*:* "*Interface Circuits for TIA/EIA-232F SLLA037A*". *[Publicado: 09/2002].*

[4] *[Publicación]* Autoria: *Texas Instruments.* Título*:* "*AN-1031 TIA/EIA-422-B Overview*". *[Publicado: 01/2000]*.

[5] *[Especificación]* Autoria: *ANSI/TIA/EIA.* Título*:* "*EIA/TIA-485-A Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems*". *[Publicado: 03/03/1998].*

[6] [*Articulo*] Autoría: *Jacob Davis.* Título: "*How to Handle Common Issues with USB to RS-232 Adapter Cables*" [*En línea*] Disponible: https://www.campbellsci.co.uk/blog/usb-rs-232-adapter-cable-issues [*Ultimo acceso*: 25/03/2023]

[7] [*Articulo*] Autoría: *Mark Casilang.* Título: "*Saleae Logic Analyzer*" [*En línea*] Disponible: https://www.jameco.com/Jameco/workshop/ProductNews/saleae-logic-analyzer.html [*Ultimo acceso*: 25/03/2023]

[8] [*Pagina web*] Autoría: *Tektronix.* Título: "*Logic Analyzer*" [*En línea*] Disponible: https://www.tek.com/en/products/logic-analyzer [*Ultimo acceso:* 25/03/2023]

[9] [*Publicación]* Autoria: *Jan Axelson.* Título*:* "*Serial Port Complete 2nd Edition*". *[Publicado: 2007].* Editorial: Lakview Reserch LLC

[10] [*Hoja de datos*] Autoría: *Prolific Technology.* Título: "*PL-2303HX Edition (Chip Rev.D) USB to Serial Brigde Controler Product Datasheet*" [*En línea*] Disponible: https://www.prolific.com.tw/UserFiles/files/ds_pl2303HXD_v1_4_4.pdf [*Ultimo acceso*: 25/03/2023]

[11] [*Hoja de datos*] Autoría: *Axis Electronics Corporation.* Titulo: " *ZT207E/ZT208E/ZT211E/ZT213E/ZT213AE Low Power 5V 250Kbps RS232 Transceivers Datasheet*" [*En línea*] Disponible: https://asix.com.tw/en/product/UART_Transceivers/5V_RS232/ZT213E [*Ultimo acceso*: 25/03/2023]

[12] [*Hoja de datos*] Autoría: *Texas Instruments.* Título: "*MAX3232 3-V to 5.5-V Multichannel RS-232 Line Driver and Receiver*" [*En línea*] Disponible: https://www.ti.com/lit/ds/symlink/max3232.pdf [*Ultimo acceso*: 25/03/2023]

[13] [*Hoja de datos*] Autoría: *Silicon Labs.* Título: "CP2102/9 Single Chip USB to UART Bridge " [*En línea*] Disponible https://www.silabs.com/documents/public/data-sheets/CP2102-9.pdf [*Ultimo acceso*: 25/03/2023]

[14] [*Hoja de datos*] Autoría: *Nanjing Qinheng Microelectronics.* Título: "USB to Serial Port Bridge CH340" [*En línea*] Disponible: http://www.wch-ic.com/downloads/CH341DS1_PDF.html [*Ultimo acceso*: 25/03/2023]

[15] [*Hoja de datos*] Autoría: *Texas Instruments.* Título: "*MAX213 5-V MULTICHANNEL RS-232 LINE DRIVER/RECEIVER*" [*En línea*] Disponible: https://www.ti.com/lit/ds/symlink/max213.pdf [*Ultimo acceso:* 25/03/2023]

[16] [*Programa Informático*] Autoría: *Simon Tatham et al.* Nombre: *PuTTY* [*En línea*] Disponible: https://www.putty.org/putty [Ú*ltimo acceso*: 25/03/2023]

[17] [*Programa Informático*] Autoría: *desconocido.* Nombre: *Realterm* [*En línea*] https://realterm.sourceforge.io/Realterm [*Último acceso*: 25/03/2023]

[18] [*Programa Informático*] Autoría: *Maettu.* Nombre: YAT [*En línea*] https://sourceforge.net/projects/y-a-terminal/files/ [*Último acceso*: 25/03/2023]

[19] [*Programa Informático*] Autoría: *Roger Meier.* Nombre: *CoolTerm* [*En línea*] https://freeware.the-meiers.org/ [*Último acceso*: 25/03/2023]

[20] [*Pagina web*] Autoría: *National Instruments.* Título: "*CompactRIO*" [*En línea*] Disponible: https://www.ni.com/es-es/shop/compactrio.html [*Ultimo acceso:* 09/04/2023]

[21] [*Pagina web*] Autoría: *National Instruments.* Título: "*NI-9871*" [*En linea*] Disponible: https://www.ni.com/es-es/shop/model/ni-9871.html [*Ultimo acceso*: 09/04/2023]

[22] [*Programa Informático*] Titulo: *Eagle*. Compañía: *Autodesk*. Disponible en: https://www.autodesk.com/products/eagle [*Ultimo acceso*: 12/03/2023].

[23] [*Programa Informático*] Titulo: *MPLAB X IDE*. Compañía: *Microchip Technologies.* Disponible en: https://www.microchip.com/en-us/tools-resources/develop/mplab-x-ide [*Ultimo acceso*: 01/04/2023].

[24] [*Publicación*] Autoría: *Microchip Technologies.* Título: "*MPLAB SNAP In-circuit Debugger User Guide*". [*Publicado:* 2022].

[25] [*Programa Informático*] Titulo: *Processing IDE*. Compañía: *Processing Foundation*. Disponible en: https://processing.org [*Ultimo acceso*: 01/05/2023].

[26] [*Pagina Web*] Autoría: *Microchip Technologies.* Título: "*PIC32MX Family of Microcontrolers*". Disponible: https://www.microchip.com/en-

us/products/microcontrollers-and-microprocessors/32-bit-mcus/pic32-32-bit-mcus/pic32mx [*Último acceso*: 03/04/2023].

[27] [*Publicación*] Autoría: *Microchip Technologies*. Título: “*PIC32MX Family Reference Manual Section 21 UART*”. [*Publicado:* 2017].

[28] [*Publicación*] Autoría: *Microchip Technologies.* Título: “*PIC32MX3XX/4XX Datasheet*”. [*Publicado*: 2011].

[29] [*Pagina web*] Autoría: *Microchip Technologies*. Título: "*Curiosity PIC32MX470 Development board*" [*En línea*] Disponible: https://www.microchip.com/en-us/development-tool/dm320103 [*Último acceso*: 25/03/2023].

[30] [*Pagina web*] Autoría: *Microchip Tech*. Título: "PIC32 USB STARTER KIT III" [*En línea*] Disponible: https://www.microchip.com/en-us/development-tool/dm320103 [*Ultimo acceso:* 25/03/2023].

[31] [*Pagina web*] Autoría: *Olimex*. Título: "*PIC32-Pinguino*" [*En línea*] Disponible: https://www.olimex.com/Products/Duino/PIC32/PIC32-PINGUINO/open-source-hardware [*Último acceso*: 25/03/2023].

[32] [*Pagina web*] Autoría: *Mikroe*. Título: "*CLICKER 2 FOR PIC32"* [*En línea*] Disponible: https://www.mikroe.com/clicker-2-pic32mx [*Ultimo acceso:* 20/03/2023].

[33] [*Pagina web*] Autoría: *Microchip Tech*. Título: "*CHIPKIT FUBARINO*" [*En línea*] Disponible: https://www.microchip.com/en-us/development-tool/TCHIP011 [*Último acceso:* 20/03/2023].

[34] [*Pagina web*] Autoría: *Northwestern*. Título: "*NU32*" [En línea] Disponible: https://hades.mech.northwestern.edu/index.php/NU32 [*Último acceso*: 20/03/2023].

[35] [*Hoja de datos*] Autoría: *Texas Instruments*. Título: "*MAX3232 3-V to 5.5-V Multichannel RS-232 Line Driver and Receiver*" [*En línea*] Disponible: https://www.ti.com/lit/ds/symlink/max3232.pdf [*Ultimo acceso*: 25/03/2023]

[36] [*Hoja de datos*] Autoría: *Texas Instruments.* Título: "*SN75C3232 3-V to 5.5-V Multichannel RS-232 Compatible Line Driver and Receiver*" [*En línea*] Disponible: https://www.ti.com/lit/ds/symlink/sn75c3232.pdf [*Ultimo acceso*: 26/04/2023].

[37] [*Hoja de datos*] Autoría: *Analog Devices*. Titulo: "*3.3V-Powered, 10Mbps and Slew-Rate-Limited True RS-485/RS-422 Transceivers*" [*En línea*] Disponible: https://www.analog.com/media/en/technical-documentation/data-sheets/max3483-max3491.pdf [*Ultimo acceso:* 26/04/2023].

[38] [*Hoja de datos*] Autoría: *Littlefuse*. Título: " *TVS Diodes SMAJ Datasheet*" [*En línea*] Disponible: https://www.littelfuse.com/products/tvs-diodes/surface-mount/smaj/smaj7_0ca.aspx [*Ultimo acceso*: 26/04/2023].

[39] [*Nota de aplicación*] Autoría: *Analog Devices by Hein Marais*. Título: "*AN-960 RS-485/RS-422 Circuit Implementation Guide*" [*En línea*] Disponible: https://www.analog.com/media/en/technical-documentation/application-notes/an-960.pdf [*Ultimo acceso:* 26/04/2023].

[40] [*Hoja de datos*] Autoría: *Waveshare.* Título: "*Waveshare LCD1602 Datasheet*" [*En línea*] Disponible: https://www.waveshare.com/datasheet/LCD_en_PDF/LCD1602.pdf [*Ultimo acceso*: 26/04/2023].

[41] [*Hoja de datos*] Autoría: *Texas Instruments*. Título: " *MC3x063A 1.5-A Peak Boost/Buck/Inverting Switching Regulators* " [*En línea*] Disponible: https://www.ti.com/lit/ds/symlink/mc34063a.pdf [*Ultimo acceso*: 26/04/2023].

[42] [*Hoja de datos*] Autoría: *Microchip Technologies*. Título: " MCP1700 *Low Quiescent Current LDO* " [*En línea*] Disponible:

https://ww1.microchip.com/downloads/en/DeviceDoc/MCP1700-Low-Quiescent-Current-LDO-20001826E.pdf [*Ultimo acceso*: 26/04/2023].

[43] [*Pagina web*] Autoría: *Digikey* Título: "*PCB trace width calculator*" [*En línea*] Disponible: https://www.digikey.es/en/resources/conversion-calculators/conversion-calculator-pcb-trace-width [*Ultimo acceso*: 20/03/2023].

[44] [*Publicación*] Autoría: *Texas Instruments*. Título: "*PCB Design Guidelines For Reduced EMI*". [*Publicado*: 10/1999]. [*En línea*] Disponible: https://www.ti.com/lit/an/szza009/szza009.pdf [*Último acceso*: 05/04/2023].

[45] [*Publicación*] Autoría: *Microchip Technologies.* Título: "*PIC32MX Family Reference Manual Section 6 Oscillators*". [Publicado: 2017].

[46] [*Librerías*] Titulo: *Microchip Libraries for Applications. Compañia : Microchip Technologies.* Disponible en: *https://www.microchip.com/en-us/tools-resources/develop/libraries/microchip-libraries-for-applications* [*Ultimo acceso:* 12/03/2023].

[47] [*Publicación*] Autoría: *Microchip Technologie*s. Título: "*AN1164 USB CDC Class on an Embedded Device* ". [*Publicado:* 2008].

[48] [*Publicación*] Autoría: *Microchip Technologies.* Título: "*PIC32MX Family Reference Manual Section 5 Flash Programming*". [*Publicado:* 2016].

[49] [*Librerías*] Titulo: *G4P*. Autor: *Peter Lager*. Disponible en: http://www.lagers.org.uk/g4p/ [*Ultimo acceso*: 12/04/2023].

[50] [*Librerías*] Titulo: *ControlP5*. Autor: *Andreas Schlegel*. Disponible en: https://www.sojamo.de/libraries/controlP5/ [*Ultimo acceso*: 12/04/2023].

8. Anexos

Anexo I: Esquemático del circuito diseñado

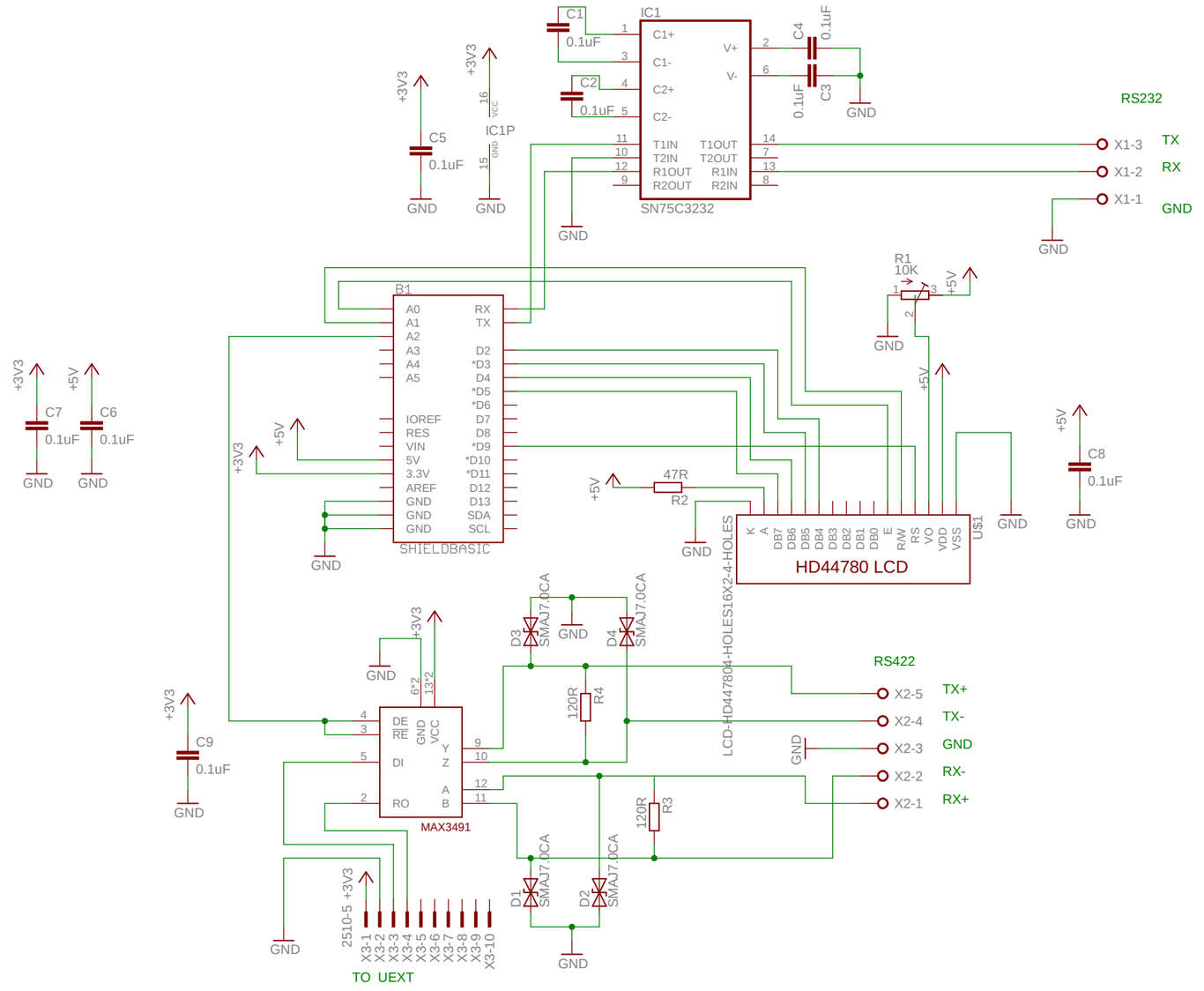
Anexo II: Diseño de PCB

Anexo III: Descripción de mensajes de configuración

Anexo IV: Código fuente del microcontrolador PIC32MX

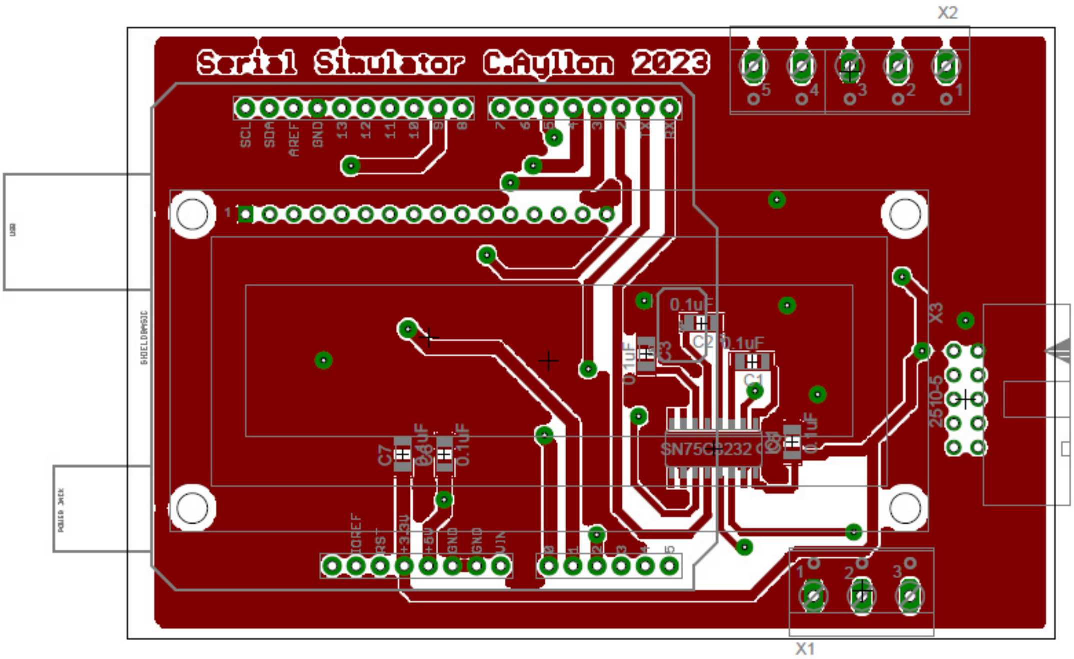
Anexo V: Esquemático de la placa PIC32-Pinguino

ANEXO I

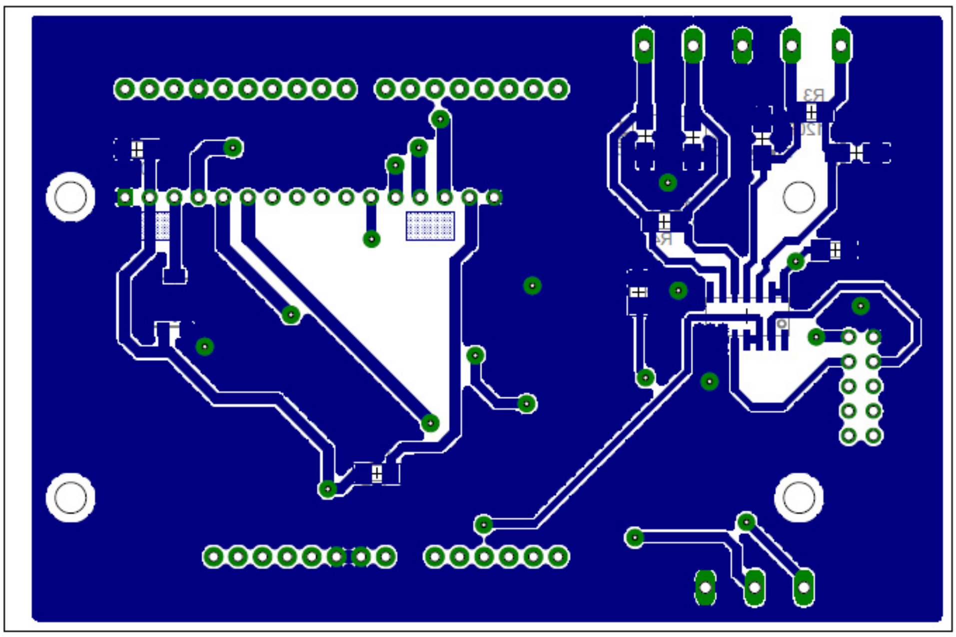


SIMULADOR SERIE RS232-RS422	DRAW BY: Carlos Aylon Fernandez
	Project: TFG UOC 2023
	28 Abril 2023
	Sheet: 1/1

ANEXO II



TOP



BOTTOM

ANEXO III

Estructura de mensajes de configuración

La configuración del simulador se envía desde el PC mediante un mensaje hexadecimal de 4 Bytes seguidos, tales como *AA A4 A1 E1* :
Donde el byte 3 ó más significativo (*AA*) configura el tipo de puerto a usar y si habrá o no contador de mensajes, con las siguientes posibilidades:

AA → Puerto RS232 sin contador de mensajes
BA → Puerto RS232 con contador de mensajes
AB → Puerto RS422 sin contador de mensajes
BB → Puerto RS422 con contador de mensajes

El contador es un primer byte incremental que se envía con cada mensaje y cuenta de *0* a *FF* y vuelve a 0.

El byte 2 del mensaje de configuración define el número bits, paridad y bits de paro donde:

A4 →8-E-1 ---------- 8 bits-paridad par-1 bit stop
B4 →8-E-2
A5 →8-O-1
B5 →8-O-2
A6 →8-N-1
B6 →8-N-2
A7 →9-N-1
B7 →9-N-2

El byte 1 define el gap de separación entre mensajes con la siguiente codificación:

A1 →10us
A2 →20us
A3 →50us
A4 →100us
A5 →200us

A6 →500us
A7 →1ms
A8 →1,5ms
A9 →2ms
AA →5ms
AB →10ms

El byte 0 define la velocidad de bits o baudrate de la transmisión:

E0 →1Mbps
E1 →937500bps
E2 →576000bps
E3 →460800bps
E4 →230400bps
E5 →115200bps
E6 →57600bps
E7 →38400bps
E8 →19200bps
E9 →9600bps
EA →4800bps
EB →2400bps

El mensaje a simular se envía codificado en ASCII precedido del símbolo *Ð* (*D0* en hex) por lo tanto solo se pueden enviar caracteres ASCII de 0 a 9 y de A a la F en mayúsculas hasta un máximo de 1024 bytes hex (2048 ASCII). Ejemplo de mensaje a enviar:
ÐA0A1A2A3FFF1BA07CAFE12CAD1 → 0xA0A1A2A3FFF1BA07CAFE12CAD1

ANEXO IV

Codigo fuente para PIC32MX

ARCHIVO MAIN.C

```
/********************************************************************
 FileName:      main.c
 Dependencies:  See INCLUDES section
 Processor:     PIC32MX USB Microcontrollers
 Hardware:      Adapted for PIC32440MX
 Complier:      Microchip XC32 (for PIC32)

 ********************************************************************
 File Description:
SIMULADOR RS232 y RS422 para PIC32MX
Carlos Ayllon TFG 2023
 ********************************************************************/

/** INCLUDES *******************************************************/
#include <xc.h>
#include <stdlib.h>
#include <stdio.h>
#include <plib.h>
#include <p32xxxx.h>
#include "./USB/usb_function_cdc.h"
#include "./RS232_Simulator.X/lcd.h"
#include "HardwareProfile.h"
#include "GenericTypeDefs.h"
#include "Compiler.h"
#include "usb_config.h"
#include "USB/usb_device.h"
#include "USB/usb.h"

/******* CONFIGURATION *********************************************/
#if defined(PIC32_USB_STARTER_KIT)
    #pragma config UPLLEN   = ON            // USB PLL Enabled
    #pragma config FPLLMUL  = MUL_15        // PLL Multiplier 60mhz
    #pragma config UPLLIDIV = DIV_2         // USB PLL Input Divider
    #pragma config FPLLIDIV = DIV_2         // PLL Input Divider
    #pragma config FPLLODIV = DIV_1         // PLL Output Divider
    #pragma config FPBDIV   = DIV_1         // Peripheral Clock divisor
    #pragma config FWDTEN   = OFF           // Watchdog Timer
    #pragma config WDTPS    = PS1           // Watchdog Timer Postscale
    #pragma config FCKSM    = CSDCMD        // Clock Switching & Fail Safe Clock Monitor
    #pragma config OSCIOFNC = OFF           // CLKO Enable
    #pragma config POSCMOD  = XT            // Primary Oscillator
    #pragma config IESO     = OFF           // Internal/External Switch-over
    #pragma config FSOSCEN  = OFF           // Secondary Oscillator Enable (KLO was off)
    #pragma config FNOSC    = PRIPLL        // Oscillator Selection
    #pragma config CP       = OFF           // Code Protect
    #pragma config BWP      = OFF           // Boot Flash Write Protect
    #pragma config PWP      = OFF           // Program Flash Write Protect
    #pragma config ICESEL   = ICS_PGx2      // ICE/ICD Comm Channel Select

#else
    #error No hardware board defined, see "HardwareProfile.h" and __FILE__
#endif


/** V A R I A B L E S ********************************************************/
```

```
char USB_In_Buffer[64];      // Bufer de PIC a PC_USB
char USB_Out_Buffer[64];     // BUfer de salida del pc USB->Pic
char USB_temp_Buffer[2052]; // bufer para meter mensajes Ascii largos en bloques de 64 char
UINT Uart_Msg_Hex[256];      // Bufer convertidor de Ascii a int32
BYTE Uart_byte_Hex[1024];
BYTE FlagConf = 0;
BYTE SetIniConf = 0;
UINT count =0;
BYTE msg_num =0;
BOOL msg_num_on=0;
BOOL port_422 = FALSE;
UINT page=0;
UINT gap_delay=0;
BYTE baud_conf_byte=0;
UINT size_msg = 0 ;
DWORD dato= 0;
DWORD baud_conf= 0;
BYTE Delay_msg = 0;
//BYTE debug=0;
BOOL stringPrinted;
volatile BOOL buttonPressed;
volatile BYTE buttonCount;
UINT Baudrate = 0;
UINT baudrates[12]={14,15,25,32,64,129,259,390,780,1561,3124,6249}; // baud to UxBRG register nº
char
*Baud_cfg[12]={"1Mb","937.5K","576K","460.8K","230.8K","115.2K","57.6Kb","38.4K","19.2K
","9600b","4800b","2400b"}; //array display lcd baud


/** P R I V A T E  P R O T O T Y P E S ***************************************/
static void InitializeSystem(void);
void ProcessIO(void);
void USBDeviceTasks(void);
void YourHighPriorityISRCode();
void YourLowPriorityISRCode();
void USBCBSendResume(void);
void BlinkUSBStatus(void);
void UserInit(void);
UINT* hex_decode(const char *in, size_t len, UINT *out);
void SetConfig(void);
void ApplyConfig(void);
void PrintU1cfg(void);
void PrintU2cfg(void);
void gap_data (BYTE gap);
void Send_message(void);
/** VECTOR REMAPPING ********************************************/

/*******************************************/

//Dirección Memoria de Programa donde se guarda la configuración
//Asegurarse de que es región de memoria libre segun micro

#define NVM_PROGRAM_PAGE 0x9D03C000 // En el Execution Memory es 1D03_C000, son paginas de 0x1000 hasta 7FFF
// Códigos  de datos para FLASH NVM
#define CODE_OK 0x60606060   // indica msg config OK en NVM
#define CODE_MSG1 ((char)0xD0)   // Ð 0xD0 msg to send
#define CODE_CONF232 ((char)0xAA )    // ª 0xAA config to apply 232
#define CODE_CONF422 ((char)0xAB )    // ª 0xAb config to apply 422
#define CODE_CONF232_C ((char)0xBA )   // 232 with counter
#define CODE_CONF422_C ((char)0xBB )    //  422 with counter
#define NVM_PAGE_SIZE   4096   // La flash se borra por paginas completas 4096 segun datasheet
```

```
/*********************************************************************
* Function:       void main(void)
*
*
* Overview:       Main program entry point.
*
* Note:           None
*********************************************************************/

int main(void)                  //*****////##### FUNCION MAIN ######""@@@@@*****

{
    InitializeSystem();
    LCD_Init();
    LCD_SetPosition(LINE_ONE);
    printf("Serial Simulator");
    // LCD_SetPosition(LINE_TWO);
    // printf("TEST DE DISPLAY");
    TRISBbits.TRISB3=0;     // Driver enable Rs422 out
    LATBbits.LATB3 = 1;     // Driver enable Rs422 on
    TRISDbits.TRISD1=0;     //led
    LATDbits.LATD1 = 1;
    DelayMs(10);
   if(PORTDbits.RD0==0){    // Si se pulsa el swicht durante el arranque se borra la configuración
       NVMErasePage((void *)NVM_PROGRAM_PAGE);
       NVMErasePage((void *)NVM_PROGRAM_PAGE+ (NVM_PAGE_SIZE*2));
       LCD_SetPosition(LINE_TWO);
    printf("DELETING CFG...");
    DelayMs(1000);
   }

    while(1)
    {
        #if defined(USB_INTERRUPT)
            if(USB_BUS_SENSE && (USBGetDeviceState() == DETACHED_STATE))
            {
                USBDeviceAttach();
            }
        #endif


        if(PORTDbits.RD0==0){   // si una vez configurado se pulsa el swicht entra en modo config USB de nuevo.
            SetIniConf=0;

        }
         if (SetIniConf==0) {
         ProcessIO();          // Proceso de entrada USB

            if(USBDeviceState == CONFIGURED_STATE){   // USB=1
              SetIniConf=0;
              LCD_WriteCmd(BLINK_OFF);
              LCD_SetPosition(LINE_TWO);

              printf(" USB Connected  ");
              SetConfig();   // recibe config por USB y lo escribe en eeprom
            }
             else if (SetIniConf==0) {
                    ApplyConfig();   // Aplica la configuracion de eeprom
             }
         }
```

```
          if (SetIniConf==1){     // Si está configurado procede a transmitir en bucle

              if (FlagConf==0b11){
                 LCD_SetPosition(LINE_ONE);
                 LCD_WriteCmd(BLINK_ON);
                 printf(" Transmiting... ");
                 LCD_WriteCmd(SHIFT_CUR_LEFT);
                 FlagConf=0;   //lcd show configured
                 }
              Send_message(); // Send message to selected port

          }
          else {
            while (!UARTTransmitterIsReady(UART1));
            WriteUART1(0xaa);
            //while(BusyUART1());
            while (!UARTTransmissionHasCompleted(UART1));

          }
      }//end while
}//end main


/*********************************************************************
 * Function:       static void InitializeSystem(void)
 *
 * PreCondition:    None
 *
 * Input:          None
 *
 * Output:         None
 *
 * Side Effects:    None
 *
 * Overview:       InitializeSystem is a centralize initialization
 *            routine. All required USB initialization routines
 *            are called from here.
 *
 *            User application initialization routine should
 *            also be called from here.
 *
 * Note:           None
 ********************************************************************/
static void InitializeSystem(void)
{

    #if defined(__C32__)
        AD1PCFG = 0xFFFF;
    #endif




    #if defined(__32MX440F256H__)|| defined(__32MX795F512H__)

    // Configure the PIC32 core for the best performance
    // at the operating frequency. The operating frequency is already set to
    // 60MHz through Device Config Registers
        SYSTEMConfigPerformance(60000000); //clock osc cpu

      #endif
```

```
//        The USB specifications require that USB peripheral devices must never source
//        current onto the Vbus pin.  Additionally, USB peripherals should not source
//        current on D+ or D- when the host/hub is not actively powering the Vbus line.
//        When designing a self powered (as opposed to bus powered) USB peripheral
//        device, the firmware should make sure not to turn on the USB module and D+
//        or D- pull up resistor unless Vbus is actively powered.  Therefore, the
//        firmware needs some means to detect when Vbus is being powered by the host.
//        A 5V tolerant I/O pin can be connected to Vbus (through a resistor), and
//        can be used to detect when Vbus is high (host actively powering), or low
//        (host is shut down or otherwise not supplying power).  The USB firmware
//        can then periodically poll this I/O pin to know when it is okay to turn on
//        the USB module/D+/D- pull up resistor.  When designing a purely bus powered
//        peripheral device, it is not possible to source current on D+ or D- when the
//        host is not actively providing power on Vbus. Therefore, implementing this
//        bus sense feature is optional.  This firmware can be made to use this bus
//        sense feature by making sure "USE_USB_BUS_SENSE_IO" has been defined in the
//        HardwareProfile.h file.
    #if defined(USE_USB_BUS_SENSE_IO)
    tris_usb_bus_sense = INPUT_PIN; // See HardwareProfile.h
    #endif

//        If the host PC sends a GetStatus (device) request, the firmware must respond
//        and let the host know if the USB peripheral device is currently bus powered
//        or self powered.  See chapter 9 in the official USB specifications for details
//        regarding this request.  If the peripheral device is capable of being both
//        self and bus powered, it should not return a hard coded value for this request.
//        Instead, firmware should check if it is currently self or bus powered, and
//        respond accordingly.  If the hardware has been configured like demonstrated
//        on the PICDEM FS USB Demo Board, an I/O pin can be polled to determine the
//        currently selected power source.  On the PICDEM FS USB Demo Board, "RA2"
//        is used for          this purpose.  If using this feature, make sure "USE_SELF_POWER_SENSE_IO"
//        has been defined in HardwareProfile - (platform).h, and that an appropriate I/O pin
//  has been mapped          to it.
    #if defined(USE_SELF_POWER_SENSE_IO)
    tris_self_power = INPUT_PIN;     // See HardwareProfile.h
    #endif

    UserInit();

    USBDeviceInit();      //usb_device.c.  Initializes USB module SFRs and firmware
                                //variables to known states.


}//end InitializeSystem



/********************************************************************
 * Function:        void UserInit(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        This routine should take care of all of code
 *                  initialization that is required.
 *
```

```
 * Note:
 *
 *******************************************************************/
void UserInit(void)
{
     //Initialize all of the debouncing variables
     buttonCount = 0;
     buttonPressed = FALSE;
     stringPrinted = TRUE;

     //Initialize all of the LED pins
       mInitAllLEDs();

     //Initialize the pushbuttons
     mInitAllSwitches();

     // Inicializa UARTs a 9600bps
     OpenUART1(UART_EN | UART_BRGH_FOUR | UART_EVEN_PAR_8BIT | UART_1STOPBIT,
UART_TX_ENABLE, 1561);
     OpenUART2(UART_EN | UART_BRGH_FOUR | UART_EVEN_PAR_8BIT | UART_1STOPBIT,
UART_TX_ENABLE, 1561);

}//end UserInit

/******************************************************************
 * Function:       void ProcessIO(void)
 *
 * PreCondition:    None
 *
 * Input:        None
 *
 * Output:        None
 *
 * Side Effects:    None
 *
 * Overview:       This function is a place holder for other user
 *           routines. It is a mixture of both USB and
 *           non-USB tasks.
 *
 * Note:         None
 *******************************************************************/
void ProcessIO(void)
{
     BYTE numBytesRead;

     //Blink the LEDs according to the USB device status
     BlinkUSBStatus();
     // User Application USB tasks
     if((USBDeviceState < CONFIGURED_STATE)||(USBSuspendControl==1)) return;

     if(buttonPressed)
     {
         if(stringPrinted == FALSE)
         {
             if(mUSBUSARTIsTxTrfReady())
             {
                 putrsUSBUSART("Button Pressed -- \r\n");
                 stringPrinted = TRUE;
             }
         }
     }
     else
     {
```

```
        stringPrinted = FALSE;
    }

    if(USBUSARTIsTxTrfReady())
    {
            numBytesRead = getsUSBUSART(USB_Out_Buffer,64);
            if(numBytesRead != 0)
            {
                  BYTE i;
            page=0;
            page++;
                  for(i=0;i<numBytesRead;i++)
                  {

                        switch(USB_Out_Buffer[i])
                        {
                              case 0x0A:   // for debug
                        // FlagConf=1;

                              case 0x0D: //for debug
                                    USB_In_Buffer[i] = USB_Out_Buffer[i];
                                    WriteUART1 (0xCA);

                                    break;
                              default:
                         USB_In_Buffer[i] = USB_Out_Buffer[i];
                                    if (numBytesRead<=64 && count<=64){
                               count++;
                         }
                         if (numBytesRead==64 && count>63){
                              USB_temp_Buffer[count-1]= USB_Out_Buffer[i];
                              count++;
                         }
                         if (numBytesRead<64 && count>63){
                              USB_temp_Buffer[count-1]= USB_Out_Buffer[i];
                              count++;
                         }

                                    break;

                        }

                  }

            putUSBUSART(USB_In_Buffer,numBytesRead);
            }
      }

    CDCTxService();


}           //end ProcessIO

/******************************************************************
 * Function:       void BlinkUSBStatus(void)
 *
 * PreCondition:   None
 *
 * Input:          None
 *
 * Output:         None
 *
 * Side Effects:   None
```

```
 *
 * Overview:        BlinkUSBStatus turns on and off LEDs
 *                  corresponding to the USB device state.
 *
 * Note:            mLED macros can be found in HardwareProfile.h
 *                  USBDeviceState is declared and updated in
 *                  usb_device.c.
 ********************************************************************/
void BlinkUSBStatus(void)
{
    static WORD led_count=0;

    if(led_count == 0)led_count = 10000U;
    led_count--;

    #define mLED_Both_Off()         {mLED_1_Off();mLED_2_Off();}
    #define mLED_Both_On()          {mLED_1_On();mLED_2_On();}
    #define mLED_Only_1_On()        {mLED_1_On();mLED_2_Off();}
    #define mLED_Only_2_On()        {mLED_1_Off();mLED_2_On();}

    if(USBSuspendControl == 1)
    {
        if(led_count==0)
        {
            mLED_1_Toggle();
            if(mGetLED_1())
            {
                mLED_2_On();
            }
            else
            {
                mLED_2_Off();
            }
        }//end if
    }
    else
    {
        if(USBDeviceState == DETACHED_STATE)
        {
            mLED_Both_Off();
        }
        else if(USBDeviceState == ATTACHED_STATE)
        {
            mLED_Both_On();
        }
        else if(USBDeviceState == POWERED_STATE)
        {
            mLED_Only_1_On();
        }
        else if(USBDeviceState == DEFAULT_STATE)
        {
            mLED_Only_2_On();
        }
        else if(USBDeviceState == ADDRESS_STATE)
        {
            if(led_count == 0)
            {
                mLED_1_Toggle();
                mLED_2_Off();
            }//end if
        }
        else if(USBDeviceState == CONFIGURED_STATE)
        {
```

```
            if(led_count==0)
            {
                 mLED_1_Toggle();
                if(mGetLED_1())
                {
                    mLED_2_Off();
                }
                else
                {
                    mLED_2_On();
                }
            }//end if
        }//end if(...)
    }//end if(UCONbits.SUSPND...)

}//end BlinkUSBStatus



// ******************************************************************************************************
// ************** USB Callback Functions ****************************************************************
// ******************************************************************************************************
// The USB firmware stack will call the callback functions USBCBxxx() in response to certain USB related
// events.  For example, if the host PC is powering down, it will stop sending out Start of Frame (SOF)
// packets to your device.  In response to this, all USB devices are supposed to decrease their power
// consumption from the USB Vbus to <2.5mA* each.  The USB module detects this condition (which according
// to the USB specifications is 3+ms of no bus activity/SOF packets) and then calls the USBCBSuspend()
// function.  You should modify these callback functions to take appropriate actions for each of these
// conditions.  For example, in the USBCBSuspend(), you may wish to add code that will decrease power
// consumption from Vbus to <2.5mA (such as by clock switching, turning off LEDs, putting the
// microcontroller to sleep, etc.).  Then, in the USBCBWakeFromSuspend() function, you may then wish to
// add code that undoes the power saving things done in the USBCBSuspend() function.

// The USBCBSendResume() function is special, in that the USB stack will not automatically call this
// function.  This function is meant to be called from the application firmware instead.  See the
// additional comments near the function.

// Note *: The "usb_20.pdf" specs indicate 500uA or 2.5mA, depending upon device classification. However,
// the USB-IF has officially issued an ECN (engineering change notice) changing this to 2.5mA for all
// devices.  Make sure to re-download the latest specifications to get all of the newest ECNs.

/******************************************************************************
 * Function:        void USBCBSuspend(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        Call back that is invoked when a USB suspend is detected
 *
 * Note:            None
 *****************************************************************************/
void USBCBSuspend(void)
{
        //Example power saving code.  Insert appropriate code here for the desired
        //application behavior.  If the microcontroller will be put to sleep, a
        //process similar to that shown below may be used:
```

```
        //ConfigureIOPinsForLowPower();
        //SaveStateOfAllInterruptEnableBits();
        //DisableAllInterruptEnableBits();
        //EnableOnlyTheInterruptsWhichWillBeUsedToWakeTheMicro();//should enable at least USBActivityIF as a wake source
        //Sleep();
        //RestoreStateOfAllPreviouslySavedInterruptEnableBits();       //Preferrably, this should be done in the
USBCBWakeFromSuspend() function instead.
        //RestoreIOPinsToNormal();                                                                    //Preferrably,
this should be done in the USBCBWakeFromSuspend() function instead.

        //IMPORTANT NOTE: Do not clear the USBActivityIF (ACTVIF) bit here.  This bit is
        //cleared inside the usb_device.c file.  Clearing USBActivityIF here will cause
        //things to not work as intended.


    #if defined(__C30__) || defined __XC16__
        USBSleepOnSuspend();
    #endif
}

/******************************************************************************
 * Function:       void USBCBWakeFromSuspend(void)
 *
 * PreCondition:    None
 *
 * Input:          None
 *
 * Output:         None
 *
 * Side Effects:    None
 *
 * Overview:       The host may put USB peripheral devices in low power
 *                                      suspend mode (by "sending" 3+ms of idle).  Once in suspend
 *                                      mode, the host may wake the device back up by sending non-
 *                                      idle state signalling.
 *
 *                                      This call back is invoked when a wakeup from USB suspend
 *                                      is detected.
 *
 * Note:           None
 *****************************************************************************/
void USBCBWakeFromSuspend(void)
{
        // If clock switching or other power savings measures were taken when
        // executing the USBCBSuspend() function, now would be a good time to
        // switch back to normal full power run mode conditions.  The host allows
        // 10+ milliseconds of wakeup time, after which the device must be
        // fully back to normal, and capable of receiving and processing USB
        // packets.  In order to do this, the USB module must receive proper
        // clocking (IE: 48MHz clock must be available to SIE for full speed USB
        // operation).
        // Make sure the selected oscillator settings are consistent with USB
      // operation before returning from this function.
}

/********************************************************************
 * Function:       void USBCB_SOF_Handler(void)
 *
 * PreCondition:    None
 *
 * Input:          None
 *
 * Output:         None
 *
```

```
 * Side Effects:    None
 *
 * Overview:        The USB host sends out a SOF packet to full-speed
 *                  devices every 1 ms. This interrupt may be useful
 *                  for isochronous pipes. End designers should
 *                  implement callback routine as necessary.
 *
 * Note:            None
 *******************************************************************/
void USBCB_SOF_Handler(void)
{
    // No need to clear UIRbits.SOFIF to 0 here.
    // Callback caller is already doing that.

    //This is reverse logic since the pushbutton is active low
    if(buttonPressed == sw3)
    {
        if(buttonCount != 0)
        {
            buttonCount--;
        }
        else
        {
            //This is reverse logic since the pushbutton is active low
            buttonPressed = !sw3;

            //Wait 100ms before the next press can be generated
            buttonCount = 100;
        }
    }
    else
    {
        if(buttonCount != 0)
        {
            buttonCount--;
        }
    }
}

/********************************************************************
 * Function:        void USBCBErrorHandler(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        The purpose of this callback is mainly for
 *                  debugging during development. Check UEIR to see
 *                  which error causes the interrupt.
 *
 * Note:            None
 *******************************************************************/
void USBCBErrorHandler(void)
{
    // No need to clear UEIR to 0 here.
    // Callback caller is already doing that.

      // Typically, user firmware does not need to do anything special
      // if a USB error occurs.  For example, if the host sends an OUT
```

```
        // packet to your device, but the packet gets corrupted (ex:
        // because of a bad connection, or the user unplugs the
        // USB cable during the transmission) this will typically set
        // one or more USB error interrupt flags.  Nothing specific
        // needs to be done however, since the SIE will automatically
        // send a "NAK" packet to the host.  In response to this, the
        // host will normally retry to send the packet again, and no
        // data loss occurs.  The system will typically recover
        // automatically, without the need for application firmware
        // intervention.

        // Nevertheless, this callback function is provided, such as
        // for debugging purposes.
}


/********************************************************************
 * Function:        void USBCBCheckOtherReq(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        When SETUP packets arrive from the host, some
 *                                          firmware must process the request and respond
 *                                          appropriately to fulfill the request.  Some of
 *                                          the SETUP packets will be for standard
 *                                          USB "chapter 9" (as in, fulfilling chapter 9 of
 *                                          the official USB specifications) requests, while
 *                                          others may be specific to the USB device class
 *                                          that is being implemented.  For example, a HID
 *                                          class device needs to be able to respond to
 *                                          "GET REPORT" type of requests.  This
 *                                          is not a standard USB chapter 9 request, and
 *                                          therefore not handled by usb_device.c.  Instead
 *                                          this request should be handled by class specific
 *                                          firmware, such as that contained in usb_function_hid.c.
 *
 * Note:            None
 *******************************************************************/
void USBCBCheckOtherReq(void)
{
    USBCheckCDCRequest();
}//end


/********************************************************************
 * Function:        void USBCBStdSetDscHandler(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        The USBCBStdSetDscHandler() callback function is
 *                                          called when a SETUP, bRequest: SET_DESCRIPTOR request
 *                                          arrives.  Typically SET_DESCRIPTOR requests are
```

```
 *                                          not used in most applications, and it is
 *                                          optional to support this type of request.
 *
 * Note:            None
 *******************************************************************/
void USBCBStdSetDscHandler(void)
{
    // Must claim session ownership if supporting this request
}//end


/******************************************************************
 * Function:        void USBCBInitEP(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        This function is called when the device becomes
 *                  initialized, which occurs after the host sends a
 *                                          SET_CONFIGURATION (wValue not = 0) request.  This
 *                                          callback function should initialize the endpoints
 *                                          for the device's usage according to the current
 *                                          configuration.
 *
 * Note:            None
 *******************************************************************/
void USBCBInitEP(void)
{
    //Enable the CDC data endpoints
    CDCInitEP();
}

/******************************************************************
 * Function:        void USBCBSendResume(void)
 *
 * PreCondition:    None
 *
 * Input:           None
 *
 * Output:          None
 *
 * Side Effects:    None
 *
 * Overview:        The USB specifications allow some types of USB
 *                                          peripheral devices to wake up a host PC (such
 *                                          as if it is in a low power suspend to RAM state).
 *                                          This can be a very useful feature in some
 *                                          USB applications, such as an Infrared remote
 *                                          control    receiver.  If a user presses the "power"
 *                                          button on a remote control, it is nice that the
 *                                          IR receiver can detect this signalling, and then
 *                                          send a USB "command" to the PC to wake up.
 *
 *                                          The USBCBSendResume() "callback" function is used
 *                                          to send this special USB signalling which wakes
 *                                          up the PC.  This function may be called by
 *                                          application firmware to wake up the PC.  This
 *                                          function will only be able to wake up the host if
 *                  all of the below are true:
```

```
*
*                                       1.  The USB driver used on the host PC supports
*                                           the remote wakeup capability.
*                                       2.  The USB configuration descriptor indicates
*                                           the device is remote wakeup capable in the
*                                           bmAttributes field.
*                                       3.  The USB host PC is currently sleeping,
*                                           and has previously sent your device a SET
*                                           FEATURE setup packet which "armed" the
*                                           remote wakeup capability.
*
*           If the host has not armed the device to perform remote wakeup,
*           then this function will return without actually performing a
*           remote wakeup sequence.  This is the required behavior,
*           as a USB device that has not been armed to perform remote
*           wakeup must not drive remote wakeup signalling onto the bus;
*           doing so will cause USB compliance testing failure.
*
*                                       This callback should send a RESUME signal that
*           has the period of 1-15ms.
*
* Note:         This function does nothing and returns quickly, if the USB
*           bus and host are not in a suspended condition, or are
*           otherwise not in a remote wakeup ready state.  Therefore, it
*           is safe to optionally call this function regularly, ex:
*           anytime application stimulus occurs, as the function will
*           have no effect, until the bus really is in a state ready
*           to accept remote wakeup.
*
*           When this function executes, it may perform clock switching,
*           depending upon the application specific code in
*           USBCBWakeFromSuspend().  This is needed, since the USB
*           bus will no longer be suspended by the time this function
*           returns.  Therefore, the USB module will need to be ready
*           to receive traffic from the host.
*
*           The modifiable section in this routine may be changed
*           to meet the application needs. Current implementation
*           temporary blocks other functions from executing for a
*           period of ~3-15 ms depending on the core frequency.
*
*           According to USB 2.0 specification section 7.1.7.7,
*           "The remote wakeup device must hold the resume signaling
*           for at least 1 ms but for no more than 15 ms."
*           The idea here is to use a delay counter loop, using a
*           common value that would work over a wide range of core
*           frequencies.
*           That value selected is 1800. See table below:
*           ==========================================================
*           Core Freq(MHz)      MIP         RESUME Signal Period (ms)
*           ==========================================================
*             48            12          1.05
*              4            1           12.6
*           ==========================================================
*           * These timing could be incorrect when using code
*             optimization or extended instruction mode,
*             or when having other interrupts enabled.
*             Make sure to verify using the MPLAB SIM's Stopwatch
*             and verify the actual signal on an oscilloscope.
*******************************************************************/
void USBCBSendResume(void)
{
    static WORD delay_count;
```

```
        //First verify that the host has armed us to perform remote wakeup.
        //It does this by sending a SET_FEATURE request to enable remote wakeup,
        //usually just before the host goes to standby mode (note: it will only
        //send this SET_FEATURE request if the configuration descriptor declares
        //the device as remote wakeup capable, AND, if the feature is enabled
        //on the host (ex: on Windows based hosts, in the device manager
        //properties page for the USB device, power management tab, the
        //"Allow this device to bring the computer out of standby." checkbox
        //should be checked).
        if(USBGetRemoteWakeupStatus() == TRUE)
        {
            //Verify that the USB bus is in fact suspended, before we send
            //remote wakeup signalling.
            if(USBIsBusSuspended() == TRUE)
            {
                USBMaskInterrupts();

                //Clock switch to settings consistent with normal USB operation.
                USBCBWakeFromSuspend();
                USBSuspendControl = 0;
                USBBusIsSuspended = FALSE;   //So we don't execute this code again,
                                             //until a new suspend condition is detected.

                //Section 7.1.7.7 of the USB 2.0 specifications indicates a USB
                //device must continuously see 5ms+ of idle on the bus, before it sends
                //remote wakeup signalling.  One way to be certain that this parameter
                //gets met, is to add a 2ms+ blocking delay here (2ms plus at
                //least 3ms from bus idle to USBIsBusSuspended() == TRUE, yeilds
                //5ms+ total delay since start of idle).
                delay_count = 3600U;
                do
                {
                    delay_count--;
                }while(delay_count);

                //Now drive the resume K-state signalling onto the USB bus.
                USBResumeControl = 1;          // Start RESUME signaling
                delay_count = 1800U;           // Set RESUME line for 1-13 ms
                do
                {
                    delay_count--;
                }while(delay_count);
                USBResumeControl = 0;          //Finished driving resume signalling

                USBUnmaskInterrupts();
            }
        }
    }
```

```
/********************************************************************
 * Function:        BOOL USER_USB_CALLBACK_EVENT_HANDLER(
 *                      int event, void *pdata, WORD size)
 *
 * PreCondition:    None
 *
 * Input:           int event - the type of event
 *                  void *pdata - pointer to the event data
 *                  WORD size - size of the event data
 *
 * Output:          None
```

```
 *
 * Side Effects:    None
 *
 * Overview:        This function is called from the USB stack to
 *                  notify a user application that a USB event
 *                  occured.  This callback is in interrupt context
 *                  when the USB_INTERRUPT option is selected.
 *
 * Note:            None
 *******************************************************************/
BOOL USER_USB_CALLBACK_EVENT_HANDLER(int event, void *pdata, WORD size)
{
    switch( event )
    {
        case EVENT_TRANSFER:
            //Add application specific callback task or callback function here if desired.
            break;
        case EVENT_SOF:
            USBCB_SOF_Handler();
            break;
        case EVENT_SUSPEND:
            USBCBSuspend();
            break;
        case EVENT_RESUME:
            USBCBWakeFromSuspend();
            break;
        case EVENT_CONFIGURED:
            USBCBInitEP();
            break;
        case EVENT_SET_DESCRIPTOR:
            USBCBStdSetDscHandler();
            break;
        case EVENT_EP0_REQUEST:
            USBCBCheckOtherReq();
            break;
        case EVENT_BUS_ERROR:
            USBCBErrorHandler();
            break;
        case EVENT_TRANSFER_TERMINATED:
            //Add application specific callback task or callback function here if desired.
            //The EVENT_TRANSFER_TERMINATED event occurs when the host performs a CLEAR
            //FEATURE (endpoint halt) request on an application endpoint which was
            //previously armed (UOWN was = 1).  Here would be a good place to:
            //1.  Determine which endpoint the transaction that just got terminated was
            //    on, by checking the handle value in the *pdata.
            //2.  Re-arm the endpoint if desired (typically would be the case for OUT
            //    endpoints).
            break;
        default:
            break;
    }
    return TRUE;
}


//******** Convierte ASCII to BIN(HEX) y lo mete en 32 bits ////
UINT* hex_decode(const char *in, size_t len, UINT *out)
{
        unsigned int i, t, hn, ln, mn2, mn3, mn4, mn5, mn6, mn7;

        for (t = 0,i = 1; i < len; i+=8,++t) {

                hn = in[i] > '9' ? in[i] - 'A' + 10 : in[i] - '0';
```

```
                ln = in[i+1] > '9' ? in[i+1] - 'A' + 10 : in[i+1] - '0';
                mn2 = in[i+2] > '9' ? in[i+2] - 'A' + 10 : in[i+2] - '0';
                mn3 = in[i+3] > '9' ? in[i+3] - 'A' + 10 : in[i+3] - '0';
                mn4 = in[i+4] > '9' ? in[i+4] - 'A' + 10 : in[i+4] - '0';
                mn5 = in[i+5] > '9' ? in[i+5] - 'A' + 10 : in[i+5] - '0';
                mn6 = in[i+6] > '9' ? in[i+6] - 'A' + 10 : in[i+6] - '0';
                mn7 = in[i+7] > '9' ? in[i+7] - 'A' + 10 : in[i+7] - '0';
                out[t] = (mn6<< 28) | (mn7 << 24)| (mn4 << 20)| (mn5 << 16)|
                         (mn2 << 12)| (mn3 << 8) | (hn << 4)| ln;
        }

        return out;
}
/***************Set de la Configuracion en la NVM *****************/
void SetConfig(void){
    if (page>=1 ) {          // Se a recibido al menos una pagina en el buffer USB
            page++;


        if (page > 32) {    // 32 pag de 64bits es el tamaño max, se evalua que se ha llegado y se procesa

            switch (USB_temp_Buffer[0])    // Se evalua el primer bit del buffer usb para identificar que recibimos
            {

                case CODE_MSG1:          // se ha recibido el mensaje OK a enviar en ASCII
                    NVMErasePage((void *)NVM_PROGRAM_PAGE); // inicializa NVM borrando 4096, es la pag
0
                    hex_decode(USB_temp_Buffer,strlen(USB_temp_Buffer),Uart_Msg_Hex);
// paso de Ascii a hex
                    page=0; //inicializa para el siguiente ciclo
                    size_msg = (count-1)/2; // tamaño del msg hex en Byte menos 1byte de Id de msg y /2 pues
char ascii son 2 byte
                    count=0; //inicializa para el siguiente ciclo

                    // Write bytes starting at Address NVM_PROGRAM_PAGE
                     DelayMs(2);
                     NVMProgram (((void *)NVM_PROGRAM_PAGE), (const void*)
Uart_Msg_Hex, size_msg, (void *) 0xA0004000);
                     DelayMs(2);
                    // Verify if data matches
                    if(memcmp(Uart_Msg_Hex, (void *)NVM_PROGRAM_PAGE, size_msg)) // si
error al comparar
                    {
                        // If not indicate an error in LCD:
                        LCD_SetPosition(LINE_ONE);
                        printf("Error writing cfg");
                    }
                else {
                    NVMWriteWord((void*)(NVM_PROGRAM_PAGE + 0x420), CODE_OK); // escribe ok
en direccion 3_c420
                    NVMWriteWord((void*)(NVM_PROGRAM_PAGE + 0x424), size_msg); // escribe en
nvm tamaño de msg
                    FlagConf= FlagConf | 0b01; // MSG Config recorded
                    LCD_SetPosition(LINE_ONE);
                    printf("Message received");
                    }
                break;
                case CODE_CONF232:   // Se ha recibido la configuracion de bitrate, paridad etc
                case CODE_CONF422:
                case CODE_CONF232_C:
                case CODE_CONF422_C:
```

```
                        NVMErasePage((void *)NVM_PROGRAM_PAGE+ (NVM_PAGE_SIZE*2)); // inicializa
pag 2 NVM borrando 4096
                        size_msg = count/2;
                        page=0;
                        count=0;
                        // Write bytes starting at Row Address NVM_PROGRAM_PAGE
                        NVMProgram (((void *)NVM_PROGRAM_PAGE)+ NVM_PAGE_SIZE*2, (const
void*) USB_temp_Buffer, 4, (void *) 0xA0004000); // 0xA00xxx->RAM rom0x3e
                        DelayMs(2);
                        // Verify if data matches
                        if(memcmp(USB_temp_Buffer, (void
*)NVM_PROGRAM_PAGE+NVM_PAGE_SIZE*2, 4))
                        {
                          // If not turn lcd to indicate an error
                          LCD_SetPosition(LINE_ONE);
                          printf("Error writing cfg");
                        }
                        FlagConf = FlagConf | 0b10;   // Baudrate config recorded
                        LCD_SetPosition(LINE_ONE);
                        printf("Baudrate cfg rcv");
                        break;
                    default:   // Cualquier codigo desconocido genera error
                        LCD_WriteCmd(CLEAR_DISPLAY);
                         LCD_SetPosition(LINE_ONE);
                         printf("Unknow config!! ");
                         page=0;
                }


            }
    }
}


void _mon_putc(char data)
{
    LCD_WriteData(data);
}

void PrintU1cfg() {    // print rs232 config
    port_422=FALSE;
    FlagConf=0b11;
    SetIniConf=1;   // mark as configured msg and baudrate
    LCD_SetPosition(LINE_TWO);
    printf("232-%s-8-E-1",Baud_cfg[Baudrate]);
}
void PrintU2cfg() {   //print rs422 config
    port_422=TRUE;
    FlagConf=0b11;
    SetIniConf=1;   // mark as configured msg and baudrate
    LCD_SetPosition(LINE_TWO);
    printf("422-%s-8-E-1",Baud_cfg[Baudrate]);
}

/***************lectura y apply de la Configuracion almacenada en NVM *****************/

void ApplyConfig(void){

    DWORD* fptr = (DWORD*)(NVM_PROGRAM_PAGE + 0x420);
      unsigned short int err;
      dato = *fptr++;
    // When a record is saved to NVM, first byte is written as 0x60606060 to indicate CODE_OK
```

```
    // that a valid record was saved.
    if(dato == CODE_OK) {   // Message present in NVM
        dato = *fptr++;   // size de msg en pos 0x424
        memcpy (Uart_byte_Hex,(void *)NVM_PROGRAM_PAGE, dato); // Copia los datos a enviar a
Uart_byte_Hex
        fptr = (DWORD*)(NVM_PROGRAM_PAGE+ (NVM_PAGE_SIZE*2)); // salta a pagina 2 donde está el config
de baudrates
        baud_conf = *fptr;
        baud_conf_byte = (BYTE)baud_conf;
        if (baud_conf_byte == 0xAA || baud_conf_byte == 0xBA){ // si los LSB es AA hay baud config ok
de rs232, si BA 232 con msg counter
            if (baud_conf_byte == 0xBA){
                msg_num_on=TRUE;   // Contador de mensajes ON
            }
            else
                msg_num_on=FALSE;   // Contador de mensajes OFF
            BYTE conf_array[sizeof(DWORD)];   // Array con campos de baudr, stop etc
            memcpy (conf_array,&baud_conf,sizeof(DWORD));
            UINT i;
            Delay_msg = conf_array[2];
            gap_data(Delay_msg);
            for (i=0xE0;i<0xEc;i++){   // bucle para obtener bit rate NVM entre 0xE0 y 0xEC
                if (conf_array[3]== i){
                    Baudrate = i;
                    Baudrate = Baudrate - 224;// Baudrat es la posicion de array de baudrates[]]
                }
            }

            if (conf_array[1]== 0xA4){        // 8bit paridad par 1 stop  8-e-1
                  // do cofig 8 bit even 1 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR | UART_EVEN_PAR_8BIT |
UART_1STOPBIT, UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();   // Print config applied to LCD
                }
            else if (conf_array[1]== 0xB4){   // 8bit paridad par 2 stop
                    // do cofig 8 bit even 2 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_EVEN_PAR_8BIT |
UART_2STOPBITS, UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();
                }
            else if (conf_array[1]== 0xA5){   // 8bit paridad impar 1 stop
                    // do cofig 8 bit odd 1 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_ODD_PAR_8BIT | UART_1STOPBIT,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();
                }
            else if (conf_array[1]== 0xB5){   // 8bit impar 2 stop
                    // do cofig 8 bit odd 2 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_ODD_PAR_8BIT | UART_2STOPBITS,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();
                }
            else if (conf_array[1]== 0xA6){   // 8bit none 1 stop
                    // do cofig 8 bit none 1 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_8BIT | UART_1STOPBIT,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();
                }
            else if (conf_array[1]== 0xB6){   // 8bit none 2 stop
                    // do cofig 8 bit none 2 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_8BIT | UART_2STOPBITS,
UART_TX_ENABLE, baudrates[Baudrate]);
```

```
                PrintU1cfg();
                }
            else if (conf_array[1]== 0xA7){   // 9bit none 1 stop
                    // do cofig 9 bit none 1 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_9BIT | UART_1STOPBIT,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();
                }
            else if (conf_array[1]== 0xB7){   // 9bit none 2 stop
                    // do cofig 9 bit none 1 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_9BIT | UART_2STOPBITS,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU1cfg();
                }


            else {
                err = 1;
                LCD_SetPosition(LINE_ONE);
                printf("Invalid Config!! ");
                return;
            }

      }
     if (baud_conf_byte == 0xAB || baud_conf_byte == 0xBB){   // si los LSB es AB hay baud config ok de
rs422 y BB 422 con msg count
            if (baud_conf_byte == 0xBB){
                msg_num_on=TRUE;
            }
            else
                msg_num_on=FALSE;
             BYTE conf_array[sizeof(DWORD)];   // Array con campos de br, stop etc
             memcpy (conf_array,&baud_conf,sizeof(DWORD));
             UINT i;
             Delay_msg = conf_array[2];
             gap_data(Delay_msg);
             for (i=0xE0;i<0xEc;i++){   // bucle para obtener bit rate NVM entre 0xE0 y 0xEC
                if (conf_array[3]== i){
                    Baudrate = i;
                    Baudrate = Baudrate - 224;// Baudrat es la posicion de array de baudrates[]]
                }
            }

            if (conf_array[1]== 0xA4){        // 422 8bit par 1 stop
                  // do cofig 8 bit even 1 stop
                OpenUART2(UART_EN | UART_BRGH_FOUR | UART_EVEN_PAR_8BIT |
UART_1STOPBIT, UART_TX_ENABLE, baudrates[Baudrate]); //baudrates[Baudrate]);
                SetIniConf=1;   // mark as configured msg and baudrate
                PrintU2cfg();   // Print config applied to LCD
                }
            else if (conf_array[1]== 0xB4){   // 8bit par 2 stop
                    // do cofig 8 bit even 2 stop
                OpenUART2(UART_EN | UART_BRGH_FOUR |UART_EVEN_PAR_8BIT |
UART_2STOPBITS, UART_TX_ENABLE, baudrates[Baudrate]);
                SetIniConf=1;
                PrintU2cfg();
                }
            else if (conf_array[1]== 0xA5){   // 8bit impar 1 stop
                    // do cofig 8 bit odd 1 stop
                OpenUART2(UART_EN | UART_BRGH_FOUR |UART_ODD_PAR_8BIT | UART_1STOPBIT,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU2cfg();
```

```
                }
            else if (conf_array[1]== 0xB5){   // 8bit impar 2 stop
                    // do cofig 8 bit odd 2 stop
                OpenUART2(UART_EN | UART_BRGH_FOUR |UART_ODD_PAR_8BIT | UART_2STOPBITS,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU2cfg();
                }
            else if (conf_array[1]== 0xA6){   // 8bit none 1 stop
                    // do cofig 8 bit none 1 stop
                OpenUART2(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_8BIT | UART_1STOPBIT,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU2cfg();
                }
            else if (conf_array[1]== 0xB6){   // 8bit none 2 stop
                    // do cofig 8 bit none 2 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_8BIT | UART_2STOPBITS,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU2cfg();
                }
            else if (conf_array[1]== 0xA7){   // 9bit none 1 stop
                    // do cofig 9 bit none 1 stop
                OpenUART1(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_9BIT | UART_1STOPBIT,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU2cfg();
                }
            else if (conf_array[1]== 0xB7){   // 9bit none 2 stop
                    //do cofig 9 bit none 1 stop
                OpenUART2(UART_EN | UART_BRGH_FOUR |UART_NO_PAR_9BIT | UART_2STOPBITS,
UART_TX_ENABLE, baudrates[Baudrate]);
                PrintU2cfg();
                }

            else {
                err = 1;
                LCD_SetPosition(LINE_ONE);
                printf("Invalid Config!! ");;
                return;
            }

     }

 }
    else{
         LCD_SetPosition(LINE_TWO); // Si no hay config en NVM
         printf("No Cfg Available");
    }
}

// ******* OBTAIN GAP VALUE FROM MESSAGE *****/
void gap_data (BYTE gap)
{
       //unsigned int int_status;
    switch (gap) {

        case 0xA1:   //10us
            gap_delay=8;
            //delay_us(0);
            break;
        case 0xA2:   //20us
            //delay_us(10);
            gap_delay=18;
            break;
```

```
        case 0xA3:   //50us
            gap_delay=48;
            break;
        case 0xA4:   //100us
            //delay_us(90);
            gap_delay=98;
            break;
        case 0xA5:   //200us
            //delay_us(190);
            gap_delay=198;
            break;
        case 0xA6:   //500us
            //delay_us(490);
            gap_delay=498;
            break;
        case 0xA7:   //1ms
            //delay_us(1000);
            gap_delay=1008;
            break;
        case 0xA8:   //1.5ms
            //delay_us(1500);
            gap_delay=1500;
            break;
        case 0xA9:   //2ms
            //delay_us(2000);
            gap_delay=2000;
            break;
        case 0xAA:   //5ms
            //delay_us(5000);
            gap_delay=5000;
            break;
        case 0xAB:   //10ms
            //delay_us(10000);
            gap_delay=10000;
            break;

    }
      return;
}

////******* FUNCION EN MAIN PARA ENVIO DEL MENSAJE POR EL PUERTO CORRESPODINTE **********//////

void Send_message()   // Envia el mensaje a al puerto configurado con el gap y contador segun config
{
    int i;
        if (msg_num_on){   // Si esta activo se añade contador de mensajes de 8bit previo a cada msg
            DelayUs(1);

            if (!port_422)
                WriteUART1(msg_num++);
            else
                WriteUART2(msg_num++);
            }
            if (!port_422){
        for (i=0;i<dato;i++){

             while (!UARTTransmitterIsReady(UART1));
               WriteUART1(Uart_byte_Hex[i]);
             }
                while(BusyUART1());
                    delay_us(gap_delay);
            }
```

```
            else
                {
            for (i=0;i<dato;i++){
            while (!UARTTransmitterIsReady(UART2));
              WriteUART2(Uart_byte_Hex[i]);
            }
            while(BusyUART2());
                delay_us(gap_delay);
           }

}
//

/** EOF main.c *************************************************/
```

ARCHIVO LCD.C

```
/*
 * File:   lcd.c
 *  Adapted for PIC32 Pinguino and LCD1602
 *
 */

#include <xc.h>
#include <plib.h>
#include <stdlib.h>
#include "main.h"
#include "lcd.h"

#if (LCD_DATA_BITS == 0x0F)
  #define LCD_DATA_ON_LOW_4_BITS
#else
  #if (LCD_DATA_BITS == 0xF0)
    #define LCD_DATA_ON_HIGH_4_BITS
  #else
    #error LCD interface supports 4-bit mode only on high or low 4-bits of one port
  #endif
#endif

static unsigned char LCD_BusyBit;

static const unsigned char CGRAM_Table[] =
{
    0b10000000, /* CGRAM character 1 */
    0b10000100,
    0b10000010,
    0b10001111,
    0b10000010,
    0b10000100,
    0b10000000,
    0b10011111,

    0b10001110, /* CGRAM character 2 */
    0b10010001,
    0b10010000,
    0b10010000,
    0b10010001,
    0b10001110,
```

```
    0b10000000,
    0b10011111,

    0b10001110,  /* CGRAM character 3 */
    0b10010001,
    0b10010000,
    0b10010011,
    0b10010001,
    0b10001110,
    0b10000000,
    0b10011111,

    0b10000000,  /* CGRAM character 4 */
    0b10001110,
    0b10001010,
    0b10001010,
    0b10001110,
    0b10000000,
    0b10000000,
    0b10011111,

    0b10011110,  /* CGRAM character 5 */
    0b10010001,
    0b10010001,
    0b10011110,
    0b10010010,
    0b10010001,
    0b10000000,
    0b10011111,

    0b10001110,  /* CGRAM character 6 */
    0b10010001,
    0b10010001,
    0b10011111,
    0b10010001,
    0b10010001,
    0b10000000,
    0b10011111,

    0b10010001,  /* CGRAM character 7 */
    0b10011011,
    0b10010101,
    0b10010101,
    0b10010001,
    0b10010001,
    0b10000000,
    0b10011111,

    0b10000000,  /* CGRAM character 8 */
    0b10000100,
    0b10001000,
    0b10011110,
    0b10001000,
    0b10000100,
    0b10000000,
    0b10011111,
};

static void LCD_E_Pulse(void)
{
    E_PIN = 1;
    DelayUs_t(1);
```

```
    E_PIN = 0;
    DelayUs_t(1);
}

static void LCD_DelayPOR(void)
{
    DelayMs(15);
}

static void LCD_Delay(void)
{
    DelayMs(5);
}

static unsigned char LCD_GetByte(void)
{
    unsigned char LCD_Data;

    LCD_PORT_DIR |= LCD_DATA_BITS; /* make LCD data bits inputs */
    RW_PIN = 1;

    E_PIN = 1;
    DelayUs_t(1);
    LCD_Data = (unsigned char)(LCD_PORT_IN & LCD_DATA_BITS);
    E_PIN = 0;
    DelayUs_t(1);

    LCD_Data = (unsigned char)((LCD_Data >> 4) | (LCD_Data << 4));

    E_PIN = 1;
    DelayUs_t(1);
    LCD_Data |= (unsigned char)(LCD_PORT_IN & LCD_DATA_BITS);
    E_PIN = 0;
    DelayUs_t(1);

#ifdef LCD_DATA_ON_HIGH_4_BITS
    LCD_Data = (unsigned char)((LCD_Data >> 4) | (LCD_Data << 4));
#endif
    return LCD_Data;
}

static void LCD_PutByte(unsigned char LCD_Data)
{
    LCD_PORT_DIR &= ~LCD_DATA_BITS; /* make LCD data bits outputs */
    RW_PIN = 0;

    /* send first nibble */
    LCD_PORT_OUT &= ~LCD_DATA_BITS;
#if !defined(LCD_DATA_ON_HIGH_4_BITS)
    LCD_Data = (unsigned char)((LCD_Data >> 4) | (LCD_Data << 4));
#endif
    LCD_PORT_OUT |= (unsigned char)(LCD_Data & LCD_DATA_BITS);
    LCD_E_Pulse();

    LCD_Data = (unsigned char)((LCD_Data >> 4) | (LCD_Data << 4));
    LCD_PORT_OUT &= ~LCD_DATA_BITS;
    LCD_PORT_OUT |= (unsigned char)((LCD_Data) & LCD_DATA_BITS);
    LCD_E_Pulse();

    LCD_PORT_DIR |= LCD_DATA_BITS; /* make LCD data bits inputs */
}

static void LCD_Busy(unsigned char TransactionType)
```

```
{
    if (LCD_BusyBit)
    {
        /* When busy bit is available test it */
        unsigned char LCD_Data;

        LCD_PORT_DIR |= LCD_DATA_BITS;  /* make LCD data bits inputs */
        LCD_Data = 0;

        RS_PIN = 0;
        RW_PIN = 1;
        do
        {
            LCD_Data = LCD_GetByte();
        } while (LCD_Data & LCD_BusyBit);
    }
    else
    {
        /* When busy bit is not available do a spin wait based on transaction type */
        if(TransactionType == 0)
        {
            DelayUs_t(40);;  /* 40 microseconds worst delay case for data type transaction */
        }
        else
        {
            DelayMs(2);   /* 2 milliseconds worst case delay for command type transaction */
        }
    }
}

void LCD_Init(void)
{
    unsigned char LCD_Data;
    unsigned char Index;

    LCD_BusyBit = 0;
    LCD_PORT_DIR &= ~LCD_DATA_BITS;  /* make LCD data bits outputs */
    E_PIN_DIR = 0;                   /* make LCD Enable strobe an output */
    RW_PIN_DIR = 0;                  /* make LCD Read/Write select an output */
    RS_PIN_DIR = 0;                  /* make LCD Register select an output */
#ifdef LCD_POWER_EN_DIR
    LCD_POWER_EN_DIR = 0;            /* make LCD Power enable an output */
#endif
    E_PIN = 0;                       /* set LCD Enable strobe to not active */
    RW_PIN = 0;                      /* set LCD Read/Write select to Write */
    RS_PIN = 0;                      /* set LCD Register select to command group */
    LCD_PORT_OUT &= ~LCD_DATA_BITS;  /* set LCD data bits to zero */
#ifdef LCD_POWER_EN
    LCD_POWER_EN = 1;                /* Turn on LCD power */
#endif
    LCD_DelayPOR();                  /* wait for LCD power on to complete */

    /* Force LCD to 8-bit mode */
    LCD_PORT_OUT &= ~LCD_DATA_BITS;  /* set LCD data bits to zero */
    LCD_PORT_OUT |= (0b00110011u & LCD_DATA_BITS);
    LCD_E_Pulse();
    LCD_Delay();
    LCD_E_Pulse();
    LCD_Delay();
    LCD_E_Pulse();
    LCD_Delay();
```

```
    /* Set LCD to 4-bit mode */
    LCD_PORT_OUT &= ~LCD_DATA_BITS; /* set LCD data bits to zero */
    LCD_PORT_OUT |= (0b00100010u & LCD_DATA_BITS);
    LCD_E_Pulse();
    LCD_Delay();

    /* Initialize LCD mode */
    LCD_WriteCmd(LCD_FORMAT);
    LCD_Delay();

    /*
  * Find position of busy bit.
  * Required when using 4-bit mode.
  */
    LCD_SetPosition(LINE_ONE+1);
    LCD_Busy(0);
    RS_PIN = 0;
    LCD_Data = LCD_GetByte();

    if (LCD_Data == 0x01)
    {
        LCD_BusyBit = 0x80;
    }
    else
    {
        if (LCD_Data == 0x10)
        {
            LCD_BusyBit = 0x08;
        }
    }

    /* Turn on display, Setup cursor and blinking */
    LCD_WriteCmd(DOFF & CURSOR_OFF & BLINK_ON);
    LCD_WriteCmd(DON & CURSOR_OFF & BLINK_OFF);
    LCD_WriteCmd(CLEAR_DISPLAY);
    LCD_WriteCmd(SHIFT_CUR_LEFT);

    /* Initialize the character generator RAM */
    LCD_SetCGRamAddr(0);
    for(Index = 0; Index < sizeof(CGRAM_Table); Index++)
    {
        LCD_WriteData(CGRAM_Table[Index]);
    }

    /* Set first position on line one, left most character */
    LCD_SetPosition(LINE_ONE);
}

void LCD_SetCGRamAddr(unsigned char data)
{
    RS_PIN = 0;
    LCD_PutByte(data | 0x40u);
    LCD_Busy(0);
}

void LCD_SetPosition(unsigned char data)
{
    RS_PIN = 0;
    LCD_PutByte(data | 0x80u);
    LCD_Busy(0);
}

void LCD_WriteCmd(unsigned char data)
```

```
{
    RS_PIN = 0;
    LCD_PutByte(data);
    LCD_Busy(1);
}

void LCD_WriteData(unsigned char data)
{
    RS_PIN = 1;
    LCD_PutByte(data);
    RS_PIN = 0;
    LCD_Busy(0);
}

void LCD_WriteString(char * pString)
{
    while(*pString)
    {
        LCD_WriteData(*pString);
        pString++;
    }
}
```

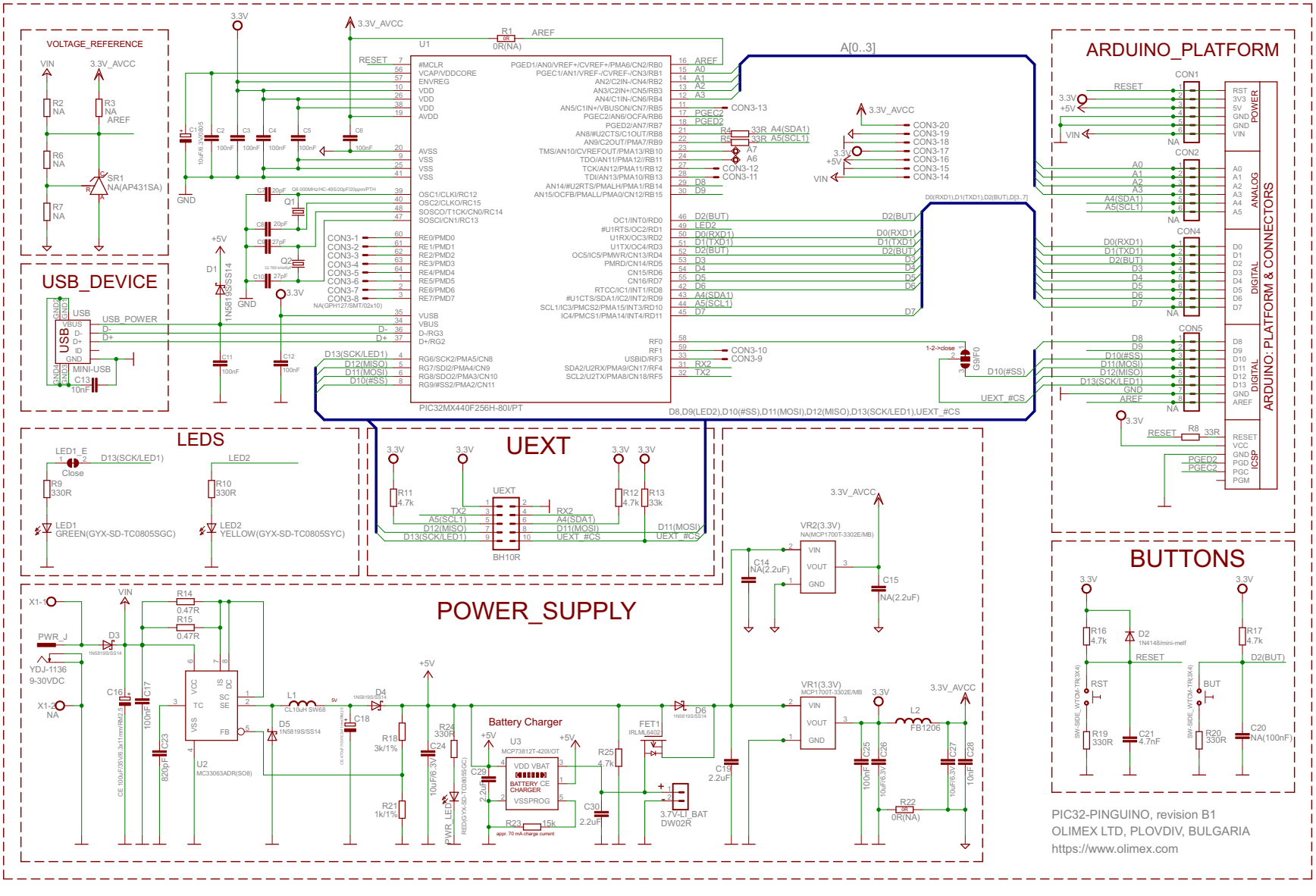
ARCHIVO Util.C

```
/*
 * File:   Util.c
 * Author: C.Ayllon
 *
 **/


#define __DELAY_C

#include "Util.h"
#define SYS_FREQ 60000000UL
#define INSTRUCTION_CLOCK_FREQUENCY 60000000UL

#if !defined(__18CXX) || defined(HI_TECH_C)
void DelayMs(WORD ms)
{
    unsigned char i;
    while(ms--)
    {
        i=4;
        while(i--)
        {
            Delay10us(25);
        }
    }
}
#endif      //#if !defined(__18CXX) || defined(HI_TECH_C)


#if defined(__C30__) || defined(__C32__)
void Delay10us(DWORD dwCount)
{
      volatile DWORD _dcnt;
```

```
	_dcnt = dwCount*((DWORD)(0.00001/(1.0/GetInstructionClock())/10));
	while(_dcnt--)
	{
		#if defined(__C32__)
			Nop();
			Nop();
			Nop();
		#endif
	}
}
//*****************************
void DelayUs(DWORD dwCount)
{
	volatile DWORD _dcnt;

	_dcnt = dwCount*((DWORD)(0.00001/(1.0/GetInstructionClock())/800));
	while(_dcnt--)
	{
		#if defined(__C32__)
			Nop();
			Nop();
			Nop();
		  Nop();
		  //Nop();

		#endif
	}
}
//*****************************
//*****************************
//******* DELAY US w/Core T*****
//*****************************
//*****************************
//Uses Core Timer
 void delay_us(unsigned int us)
{
 us *= (SYS_FREQ / 2000000 / 1); // Core timer updates every 2 ticks
 _CP0_SET_COUNT(0); // Set Core Timer count to 0
 while (us > _CP0_GET_COUNT()); // Wait until Core Timer count reaches the number we calculated earlier

}


//*****************************
//*****************************
//********** DELAY w/timer 2 ******
//*****************************
//*****************************
//Uses  Timer2
void delay_tmr2 (uint16_t delay)
{
	unsigned int int_status;

	int_status = INTDisableInterrupts();
	OpenTimer2(T2_ON | T2_SOURCE_INT | T2_PS_1_32,(1*delay));   // Core Timer updates every 2
ticks
	mT2ClearIntFlag();                                          //Clear core timer interrupt flag before re-enabling
interrupts to avoid posibility of ISR from another interrupt causing us to miss the CT interrupt.
	INTRestoreInterrupts(int_status);
	while( !mT2GetIntFlag() );
```

```
      mT2ClearIntFlag();
}

//*****************************
void Delayms (unsigned t)
    { T1CON =0x8000; // enable timer 1 tpb 1:1
    while (t--)
    {
        TMR1=0;
        while (TMR1 < 60000000L/940);
        Nop();
    }
}
//*****************************
 void DelayUs_t (unsigned int t)

    { T1CON =0x8000; // enable timer 1 tpb 1:1

    while (t--)
    {
        TMR1=0;
        while (TMR1 < 128);

    }
 }

#endif
```

ANEXO V



PIC32-PINGUINO, revision B1
OLIMEX LTD, PLOVDIV, BULGARIA
https://www.olimex.com