UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA UNIVERSITARIA DE INGENIERÍA TÉCNICA DE TELECOMUNICACIONES

PROYECTO FIN DE CARRERA

ACONDICIONAMIENTO Y PROCESADO DE SEÑALES MEDIANTE BUS CAN EN VEHÍCULOS MONOPLAZA

E.U.I.T. TELECOMUNICACIÓN

PROYECTO FIN DE CARRERA

PLAN 2000

TEMA: APLICACIONES ELECTRÓNICAS A LA FORMULA SAE

TÍTULO: ACONDICIONAMIENTO Y PROCESADO DE SEÑALES MEDIANTE BUS CAN EN VEHÍCULOS MONOPLAZA.

AUTOR: EDUARDO JOSÉ MAYORAL RUIZ

TUTOR: PEDRO COBOS ARRIBAS **VºBº.**
DEPARTAMENTO: SISTEMAS ELECTRÓNICOS DE CONTROL
Miembros del Tribunal Calificador:
PRESIDENTE: LOURDES LÓPEZ SANTIDRIÁN
VOCAL: PEDRO COBOS ARRIBAS
SECRETRARIO: JOSÉ ANTONIO HERRERA CAMACHO
DIRECTOR:

Fecha de Lectura: 7 – Septiembre – 2012
Calificación: **El Secretario,**

RESUMEN DEL PROYECTO:

En este proyecto se pretende diseñar e implementar una tarjeta que procese el control de una serie de periféricos, donde se desarrollará una interfaz de telemetría con un sistema modular de adquisición de datos mediante bus CAN (Controller Area Network).
El bus CAN ofrece sencillez debido a que es un protocolo de comunicaciones desarrollado inicialmente para aplicaciones en automóviles constando de tres capas específicas: capa física, capa de enlace de datos y capa de aplicación.
El sistema tendrá una serie de sensores que generaran la información y se transmitirán a un microcontrolador que procesará la información para posteriormente poder recoger y/o gestionar dichos datos. Por tanto, de este modo se conseguirá recoger y manejar datos en tiempo real de un vehículo, compartiéndose los datos mediante el bus CAN con otros sistemas que estén incorporados en ese vehículo.

1. RESUMEN DEL PROYECTO

En este proyecto se ha diseñado un sistema de adquisición y uso compartido de datos orientado a la implantación en un vehículo monoplaza de Formula SAE. Más concretamente, se encarga de recoger la información proporcionada por cuatro sensores infrarrojos de temperatura que sondearán constantemente la temperatura a la que se encuentran las ruedas del vehículo. La información, recogida en una memoria de almacenamiento masivo, se compartirá con otros dispositivos mediante un bus común.

Los sensores empleados para generar la información los proporciona Melexis. Dichos sensores permiten estar todos simultáneamente conectados en un bus común gracias a su electrónica interna.
Mediante el bus I^2C irán conectados los cuatro sensores de nuestra aplicación (uno por cada rueda) permitiéndose añadir a posteriori más sensores o incluso otros elementos que permitan la comunicación por este tipo de bus I^2C.

La gestión de las tareas se realiza mediante el microcontrolador DSPIC33FJ256GP710-I/PF proporcionado por Microchip. Este es un microcontrolador complejo, por lo que para nuestra aplicación desaprovecharemos parte de su potencial. En nuestra tarjeta ha sido solamente añadido el uso de los dos I^2C (uno para la tarjeta SD y el otro para los sensores), el módulo ECAN1 (para las comunicaciones por bus CAN), el módulo SPI (para acceder a una memoria Flash), 4 ADCs (para posibles mediciones) y 2 entradas de interrupción (para posible interactuación con el usuario), a parte de los recursos internos necesarios.

En este proyecto se realiza tanto el desarrollo de una tarjeta de circuito impreso dedicada a resolver la funcionalidad requerida, así como su programación a través del entorno de programación facilitado por Microchip, el ICD2 Programmer.

2. ABSTRACT

In this project, an acquisition and sharing system of data, which is oriented to be installed in a Formula SAE single-seater vehicle, has been designed. Concretely, it is responsible for getting the information supplied by four IR temperature sensors that monitor the wheels temperature. The information, which is loaded in a massive storage memory, will be shared with other devices by means of a common bus.

The sensors used to generate the information are supplied by Melexis. Such specific sensors let that all they can be connected to the same bus at the same time due to their internal electronic.
The four sensors will be connected through an I^2C bus, one for each wheel, although we could add later more sensors or even other devices that they were able to let the I^2C communication.

Tasks management will be done by means of the DSPIC33FJ256GP710-I/PF microcontroller, which will be supplied by Microchip. This is a complex microcontroller, so, in our application we waste off a part of its potential. In our PCB has only been incorporated the use of the two I^2C (one for the SD card and the other for the sensors), the ECAN module (to communicate devices), the SPI module (to access to the Flash memory), 4 ADC's (for possible measurements) and 2 interrupt inputs (for possible inter-action with the user), a part of the necessary internal resources.

This project aims the PCB development dedicated to solve the requested functionality and its programming through the programming environment provided by Microchip (the ICD2 programmer).

3. ÍNDICE

4. OBJETIVO DEL PROYECTO

Mediante los requisitos obtenidos por el usuario final de nuestro sistema, solicitan un sistema de muestreo de la temperatura de las ruedas que permita conocer los datos en tiempo real y con la capacidad de acceder a un bus CAN compartido por los demás dispositivos de control y gestión del sistema del vehículo ya implantados.
Además, aconsejan utilizar un microcontrolador de microchip.

El objetivo que se plantea es realizar una tarjeta que, basándose en la tecnología y arquitecturas que contienen los microcontroladores de Microchip, sea capaz de realizar una gestión efectiva en la captura de unos datos de temperatura y de su envío al medio compartido por el bus CAN para que el resto del sistema conozca el estado de las ruedas y pueda ajustarse y optimizar su funcionamiento, consumo y rendimiento para una mejor gestión de los recursos del monoplaza. Es decir, pretendemos retrasar y ralentizar el deterioro de los neumáticos, así como posibles problemas derivados, conociendo su estado instantáneo y ajustando lo máximo posible el funcionamiento del vehículo a sus curvas de trabajo ideales (del material que implementa físicamente las ruedas).

Todo esto, en el contexto de las carreras de coches se traduce en la obtención de resultados al adquirir ventajas de tiempo, o lo que es lo mismo, ganar segundos de ventaja por un empleo más correcto de los recursos.

Junto con otros sensores acoplados en otros puntos clave del coche, se conseguirá un preciso conocimiento del estado del vehículo en tiempo real, donde esta información se enviará a una pantalla para que la visualice el piloto y también al equipo técnico del vehículo para que pueda adoptar las medidas oportunas, llegándoles a ellos mediante ondas de radio, bluetooth, wifi u otro.

5. PROPUESTA HARDWARE

Tras analizar los requisitos solicitados, se llega a la conclusión de que lo primero que debe hacerse es empezar eligiendo un microcontrolador adecuado, que tuviese integrado el mayor número de módulos requeridos para simplificar la futura placa y la información anexa para construir e implementar dichos módulos (también para programarlos cuando estuviese producida la tarjeta).
Analizando el programa de muestras ofrecido por Microchip, se puede observar que de la gran variedad de elementos que ofrecen, eran bastantes los que integraban todos los requisitos. Finalmente se decide optar por el dsPIC33FJ256GP710 por lo interesante que resulta la idea de realizar una placa de circuito impreso (PCB) añadiendo un microcontrolador en encapsulado 100-TQFP. Además, para futuros diseños aprender a manejar tal elemento resulta muy útil debido a sus amplias capacidades.

En segunda instancia, surge la idea de introducir memorias para almacenar datos. Concretamente una Flash y otra SD. Con esto se consigue un detalle muy interesante: El tener una tarjeta móvil en tu placa que puedes extraer para analizar los datos en un PC.

Después, a la hora de elegir el programador del microcontrolador a utilizar, a parte de ponerle el programador oficial de Microchip, el MPLAB ICD2, se introduce también lo necesario para poder utilizar el programador Pickit2 de forma complementaria, ya que su precio y versatilidad suponen la posibilidad de poder programar y testear desde cualquier ordenador el dispositivo a través de USB.

También, y de cara a futuras mejoras, se decide añadir un par de entradas de interrupción y unos cuantos ADCs.

Y por último no podemos olvidar la alimentación del sistema. Teniendo en cuenta que la alimentación inicialmente se extraerá de la batería del vehículo, 12 V, serán necesarios un par de reguladores, estables y fiables, que proporcionen las tensiones necesarias en el sistema: 5 V y 3,3 V.

6. PROPUESTA SOFTWARE

Para el desarrollo de este sistema en PCB basada en un microcontrolador de Microchip, se ha utilizado un entorno de desarrollo proporcionado por la misma empresa. Dicho entorno de programación, el MPLAB IDE v8.83, va acompañado de varios depuradores In-Circuit así como de distintos simuladores y herramientas complementarias.
Es necesario añadir las librerías pertinentes relacionadas con el microcontrolador que vayamos a emplear, así como un compilador que las entienda y anexe correctamente a la hora de construir el proyecto. Por tanto, el compilador empleado será el MPLAB C30 C, un compilador de lenguaje C y C++ proporcionado también por Microchip. Cabe destacar que cualquier otro compilador, no necesariamente de Microchip, también habría servido (como por ejemplo el CSS Compiler).

Dentro de nuestro programa, se desarrolla un hilo principal que se encarga de la gestión completa de la aplicación, dividiendo en subtareas el código para simplificarlo y clarificar su lectura.
Este código empleará los recursos internos necesarios para el correcto funcionamiento de la aplicación, atendiendo al uso de “TIMERs”, “ECANs”, “SPIs”, “I2Cs”, “ADCs”, etc.

7. PRUEBAS A REALIZAR

El desarrollo de las pruebas será parcial y necesario al finalizar cada parte del proceso.
Las pruebas a realizar son:

- Prueba de continuidad en las pistas antes de la soldadura de los elementos
- Prueba visual con microscopio electrónico después de soldar el µC
- Prueba de continuidad después de la soldadura de los elementos
- Prueba de alimentación
- Prueba de funcionamiento de la tarjeta generando un pulso por un pin
- Prueba de lectura de un sensor
- Prueba de lectura de 4 sensores
- Prueba de generación de trama ECAN
- Prueba de recepción de trama ECAN por otro dispositivo

Las pruebas de funcionamiento de este proyecto se realizarán en los laboratorios de la E.U.I.T.T.
El material del que se dispone para las pruebas es:

- Tarjeta Explorer 16 Development Board
- CAN Bus Monitor Demo Board
- Osciloscopios que alcanzan anchos de banda de hasta 200MHz
- Fuentes de alimentación
- Ordenadores con sistema operativo Windows XP
- Cables de interconexión

8. DESCRIPCIÓN TÉCNICA

8.1. MICROCONTROLADOR dsPIC33fj256gp710A-I/PF

Analizando un poco el nombre del microcontrolador elegido, se puede ver que es un híbrido entre un DSP (Digital Signal Processor) y un PIC (Programable Integrated Circuit), por lo que combina un procesador de señal digital integrando periféricos internos con otras ventajas de los microcontroladores. También se observa que el "256" se refiere a la cantidad de memoria flash programable en Kbytes de la que dispone. En esta familia de microcontroladores, los que terminan en 310, 510 ó 710 como es nuestro caso, tienen 100 pines. Y por último, el término I/PF indica que el encapsulado TQFP tiene las medidas de 14x14x1 (width-length-thickness respectivamente). La otra variante para estos de 100 pines es PT (12x12x1).

Para un MCU de distinto número de pines se debe revisar el datasheet.

Características del DSPIC33:

Operating Range:

- Up to 40 MIPS operation (at 3.0-3.6V):
  - Industrial temperature range (-40°C to +85°C)

High-Performance DSC CPU:

- Modified Harvard architecture
- C compiler optimized instruction set
- 16-bit wide data path
- 24-bit wide instructions
- Linear program memory addressing up to 4M instruction words
- Linear data memory addressing up to 64 Kbytes
- 83 base instructions: mostly 1 word/1 cycle
- Sixteen 16-bit General Purpose Registers
- Two 40-bit accumulators:
  - With rounding and saturation options
- Flexible and powerful addressing modes:
  - Indirect, Modulo and Bit-Reversed
- Software stack
- 16 x 16 fractional/integer multiply operations
- 32/16 and 16/16 divide operations
- Single-cycle multiply and accumulate:
  - Accumulator write back for DSP operations
  - Dual data fetch
- Up to ±16-bit shifts for up to 40-bit data

Direct Memory Access (DMA):

- 8-channel hardware DMA:
- 2 Kbytes dual ported DMA buffer area (DMA RAM) to store data transferred via DMA:
  - Allows data transfer between RAM and a peripheral while CPU is executing code (no cycle stealing)
- Most peripherals support DMA

Interrupt Controller:

- 5-cycle latency
- Up to 63 available interrupt sources
- Up to five external interrupts
- Seven programmable priority levels
- Five processor exceptions

Digital I/O:

- Up to 85 programmable digital I/O pins
- Wake-up/Interrupt-on-Change on up to 24 pins
- Output pins can drive from 3.0V to 3.6V
- All digital input pins are 5V tolerant
- 4 mA sink on all I/O pins

On-Chip Flash and SRAM:

- Flash program memory, up to 256 Kbytes
- Data SRAM, up to 30 Kbytes (includes 2 Kbytes of DMA RAM):

System Management:

- Flexible clock options:
  - External, crystal, resonator, internal RC
  - Fully integrated PLL
  - Extremely low jitter PLL
- Power-up Timer
- Oscillator Start-up Timer/Stabilizer
- Watchdog Timer with its own RC oscillator
- Fail-Safe Clock Monitor
- Reset by multiple sources

Power Management:

- On-chip 2.5V voltage regulator
- Switch between clock sources in real time
- Idle, Sleep and Doze modes with fast wake-up

Timers/Capture/Compare/PWM:

- Timer/Counters, up to nine 16-bit timers:
  - Can pair up to make four 32-bit timers
  - 1 timer runs as Real-Time Clock with external 32.768 kHz oscillator
  - Programmable prescaler
- Input Capture (up to eight channels):
  - Capture on up, down or both edges
  - 16-bit capture input functions
  - 4-deep FIFO on each capture
- Output Compare (up to eight channels):
  - Single or Dual 16-Bit Compare mode
  - 16-bit Glitchless PWM mode

Figura 8.1.1

Communication Modules:

- 3-wire SPI (up to two modules):
  - Framing supports I/O interface to simple codecs
  - Supports 8-bit and 16-bit data
  - Supports all serial clock formats and sampling modes
- I^2C™ (up to two modules):
  - Full Multi-Master Slave mode support
  - 7-bit and 10-bit addressing
  - Bus collision detection and arbitration
  - Integrated signal conditioning
  - Slave address masking
- UART (up to two modules):
  - Interrupt on address bit detect
  - Interrupt on UART error
  - Wake-up on Start bit from Sleep mode
  - 4-character TX and RX FIFO buffers
  - LIN bus support
  - IrDA® encoding and decoding in hardware
  - High-Speed Baud mode
  - Hardware Flow Control with CTS and RTS
- Data Converter Interface (DCI) module:
  - Codec interface
  - Supports I^2S and AC'97 protocols
  - Up to 16-bit data words, up to 16 words per frame
  - 4-word deep TX and RX buffers
- Enhanced CAN (ECAN™ module) 2.0B active (up to 2 modules):
  - Up to eight transmit and up to 32 receive buffers
  - 16 receive filters and three masks
  - Loopback, Listen Only and Listen All Messages modes for diagnostics and bus monitoring
  - Wake-up on CAN message
  - Automatic processing of Remote Transmission Requests
  - FIFO mode using DMA
  - DeviceNet™ addressing support

Analog-to-Digital Converters (ADCs):

- Up to two ADC modules in a device
- 10-bit, 1.1 Msps or 12-bit, 500 ksps conversion:
  - Two, four or eight simultaneous samples
  - Up to 32 input channels with auto-scanning
  - Conversion start can be manual or synchronized with one of four trigger sources
  - Conversion possible in Sleep mode
  - ±1 LSb max integral nonlinearity
  - ±1 LSb max differential nonlinearity

CMOS Flash Technology:

- Low-power, high-speed Flash technology
- Fully static design
- 3.3V (±10%) operating voltage
- Industrial temperature
- Low-power consumption

Packaging:

- 100-pin TQFP (14x14x1 mm and 12x12x1 mm)
- 80-pin TQFP (12x12x1 mm)
- 64-pin TQFP (10x10x1 mm)

Note:	See the device variant tables for exact peripheral features per device.

Figura 8.1.2

Tabla de propósito general:

Toda esa información anterior tan detallada, se puede revisar rápidamente en la siguiente tabla:

dsPIC33F General Purpose Family Controllers

Device	Pins	Program Flash Memory (Kbyte)	RAM (Kbyte)(1)	16-bit Timer	Input Capture	Output Compare Std. PWM	Codec Interface	ADC	UART	SPI	I²C™	Enhanced CAN™	I/O Pins (Max)(2)	Packages
dsPIC33FJ64GP206	64	64	8	9	8	8	1	1 ADC, 18 ch	2	2	1	0	53	PT
dsPIC33FJ64GP306	64	64	16	9	8	8	1	1 ADC, 18 ch	2	2	2	0	53	PT
dsPIC33FJ64GP310	100	64	16	9	8	8	1	1 ADC, 32 ch	2	2	2	0	85	PF, PT
dsPIC33FJ64GP706	64	64	16	9	8	8	1	2 ADC, 18 ch	2	2	2	2	53	PT
dsPIC33FJ64GP708	80	64	16	9	8	8	1	2 ADC, 24 ch	2	2	2	2	69	PT
dsPIC33FJ64GP710	100	64	16	9	8	8	1	2 ADC, 32 ch	2	2	2	2	85	PF, PT
dsPIC33FJ128GP206	64	128	8	9	8	8	1	1 ADC, 18 ch	2	2	1	0	53	PT
dsPIC33FJ128GP306	64	128	16	9	8	8	1	1 ADC, 18 ch	2	2	2	0	53	PT
dsPIC33FJ128GP310	100	128	16	9	8	8	1	1 ADC, 32 ch	2	2	2	0	85	PF, PT
dsPIC33FJ128GP706	64	128	16	9	8	8	1	2 ADC, 18 ch	2	2	2	2	53	PT
dsPIC33FJ128GP708	80	128	16	9	8	8	1	2 ADC, 24 ch	2	2	2	2	69	PT
dsPIC33FJ128GP710	100	128	16	9	8	8	1	2 ADC, 32 ch	2	2	2	2	85	PF, PT
dsPIC33FJ256GP506	64	256	16	9	8	8	1	1 ADC, 18 ch	2	2	2	1	53	PT
dsPIC33FJ256GP510	100	256	16	9	8	8	1	1 ADC, 32 ch	2	2	2	1	85	PF, PT
dsPIC33FJ256GP710	100	256	30	9	8	8	1	2 ADC, 32 ch	2	2	2	2	85	PF, PT

Note 1: RAM size is inclusive of 2 Kbytes DMA RAM.
2: Maximum I/O pin count includes pins shared by the peripheral functions.

Figura 8.1.3

Aquí se puede comprobar rápidamente, dentro de la misma familia de controladores, que diferencias existen entre todos ellos y los elementos de los que disponen y que podrían necesitarse.
También se ofrece en el catálogo la posibilidad de ver de forma gráfica la distribución interna de esta información en el diagrama de bloques mostrado en la siguiente página.

Diagrama de bloques:

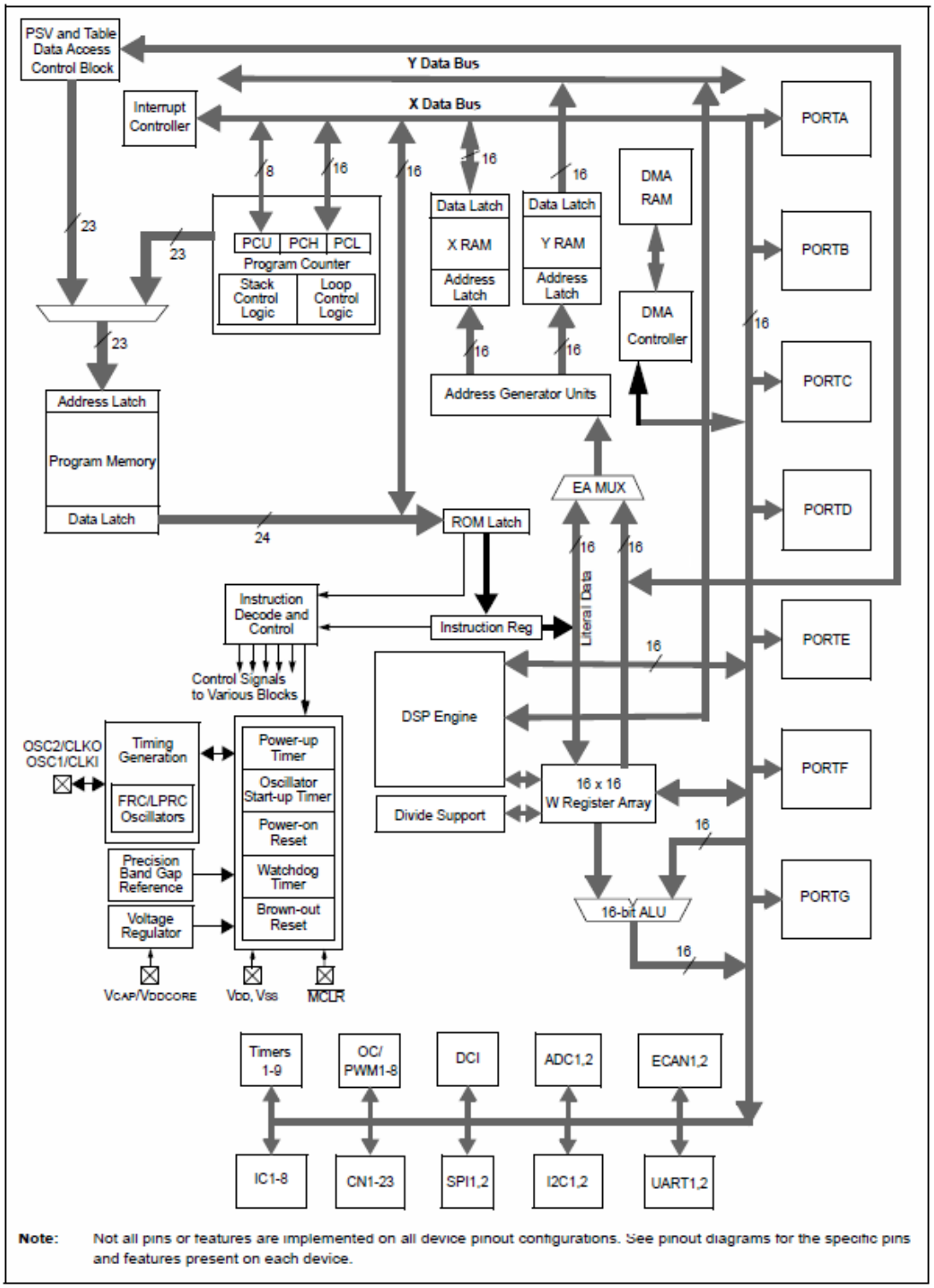


Figura 8.1.4

Por lo que llegados a este punto, ya se sabe toda la información de sobre qué elementos son de los que dispone el microcontrolador. Ahora se necesita saber como están situados y por supuesto, como se utilizan.

Diagrama de pines:

Para una familiarización más detallada con el MCU, es necesario ver en su diagrama de pines la localización física de los elementos que en las dos anteriores páginas se destacaba:

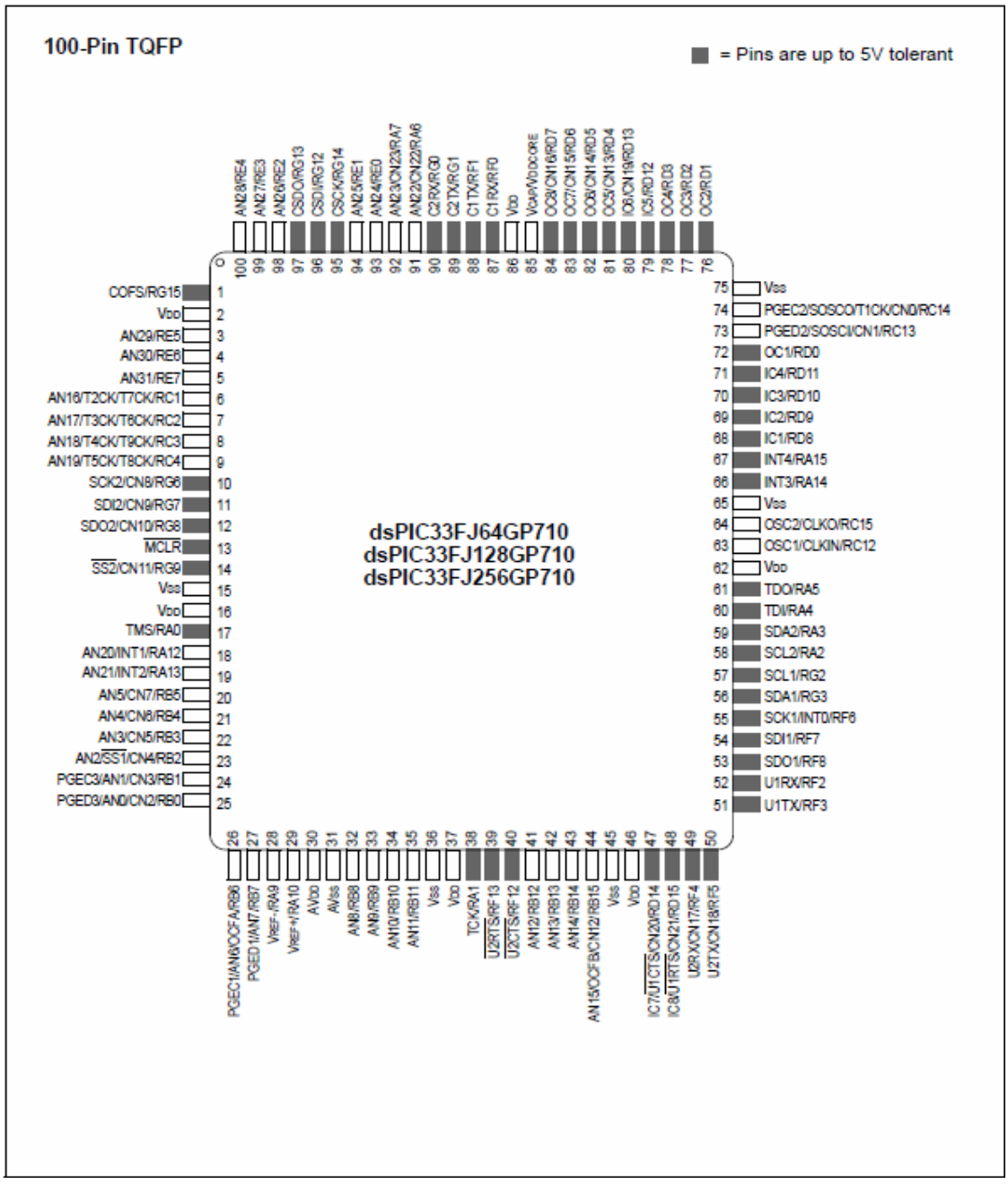


Figura 8.1.5

Por último es necesario consultar la información que el fabricante proporciona en el catálogo de la descripción de los pines.

Esto nos dará una idea más concreta de la utilidad de cada módulo y de sus pines asociados.

Pin Name	Pin Type	Buffer Type	Description
AN0-AN31	I	Analog	Analog input channels.
AVDD	P	P	Positive supply for analog modules. This pin must be connected at all times.
AVSS	P	P	Ground reference for analog modules.
CLKI CLKO	I O	ST/CMOS —	External clock source input. Always associated with OSC1 pin function. Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. Optionally functions as CLKO in RC and EC modes. Always associated with OSC2 pin function.
CN0-CN23	I	ST	Input change notification inputs. Can be software programmed for internal weak pull-ups on all inputs.
COFS CSCK CSDI CSDO	I/O I/O I O	ST ST ST —	Data Converter Interface frame synchronization pin. Data Converter Interface serial clock input/output pin. Data Converter Interface serial data input pin. Data Converter Interface serial data output pin.
C1RX C1TX C2RX C2TX	I O I O	ST — ST —	ECAN1 bus receive pin. ECAN1 bus transmit pin. ECAN2 bus receive pin. ECAN2 bus transmit pin.
PGED1 PGEC1 PGED2 PGEC2 PGED3 PGEC3	I/O I I/O I I/O I	ST ST ST ST ST ST	Data I/O pin for programming/debugging communication channel 1. Clock input pin for programming/debugging communication channel 1. Data I/O pin for programming/debugging communication channel 2. Clock input pin for programming/debugging communication channel 2. Data I/O pin for programming/debugging communication channel 3. Clock input pin for programming/debugging communication channel 3.
IC1-IC8	I	ST	Capture inputs 1 through 8.
INT0 INT1 INT2 INT3 INT4	I I I I I	ST ST ST ST ST	External interrupt 0. External interrupt 1. External interrupt 2. External interrupt 3. External interrupt 4.
MCLR	I/P	ST	Master Clear (Reset) input. This pin is an active-low Reset to the device.
OCFA OCFB OC1-OC8	I I O	ST ST —	Compare Fault A input (for Compare Channels 1, 2, 3 and 4). Compare Fault B input (for Compare Channels 5, 6, 7 and 8). Compare outputs 1 through 8.
OSC1 OSC2	I I/O	ST/CMOS —	Oscillator crystal input. ST buffer when configured in RC mode; CMOS otherwise. Oscillator crystal output. Connects to crystal or resonator in Crystal Oscillator mode. Optionally functions as CLKO in RC and EC modes.
RA0-RA7 RA9-RA10 RA12-RA15	I/O I/O I/O	ST ST ST	PORTA is a bidirectional I/O port.
RB0-RB15	I/O	ST	PORTB is a bidirectional I/O port.
RC1-RC4 RC12-RC15	I/O I/O	ST ST	PORTC is a bidirectional I/O port.
RD0-RD15	I/O	ST	PORTD is a bidirectional I/O port.
RE0-RE7	I/O	ST	PORTE is a bidirectional I/O port.
RF0-RF8 RF12-RF13	I/O I/O	ST ST	PORTF is a bidirectional I/O port.

Legend: CMOS = CMOS compatible input or output; Analog = Analog input; P = Power
ST = Schmitt Trigger input with CMOS levels; O = Output; I = Input

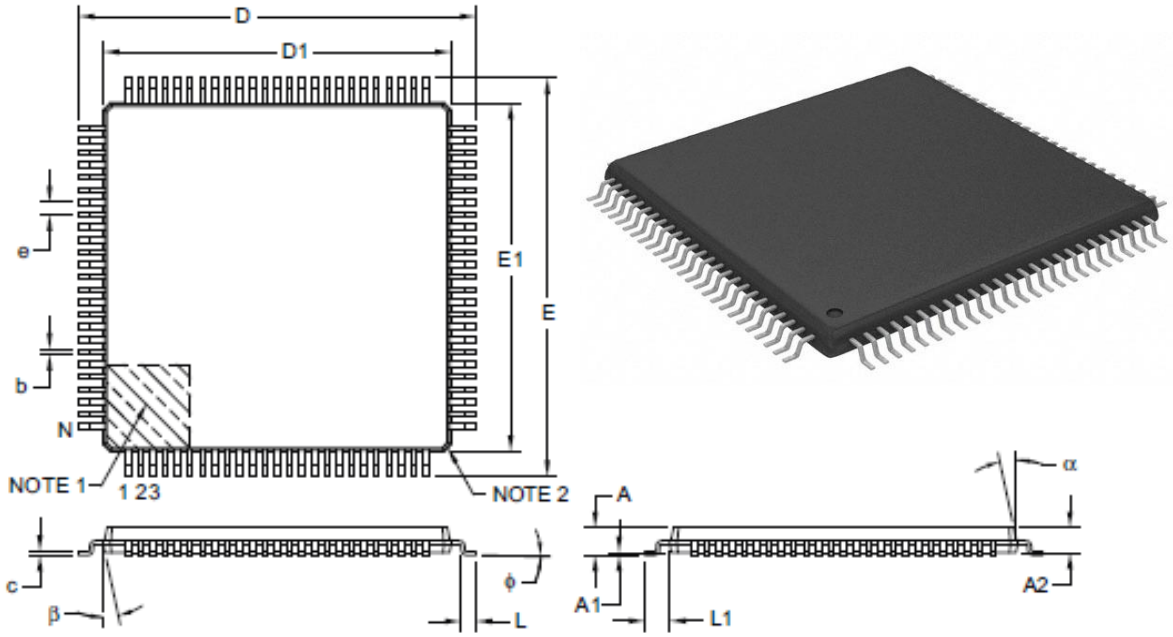
Figura 8.1.6

Pin Name	Pin Type	Buffer Type	Description
RG0-RG3	I/O	ST	PORTG is a bidirectional I/O port.
RG6-RG9	I/O	ST	
RG12-RG15	I/O	ST	
SCK1	I/O	ST	Synchronous serial clock input/output for SPI1.
SDI1	I	ST	SPI1 data in.
SDO1	O	—	SPI1 data out.
SS1	I/O	ST	SPI1 slave synchronization or frame pulse I/O.
SCK2	I/O	ST	Synchronous serial clock input/output for SPI2.
SDI2	I	ST	SPI2 data in.
SDO2	O	—	SPI2 data out.
SS2	I/O	ST	SPI2 slave synchronization or frame pulse I/O.
SCL1	I/O	ST	Synchronous serial clock input/output for I2C1.
SDA1	I/O	ST	Synchronous serial data input/output for I2C1.
SCL2	I/O	ST	Synchronous serial clock input/output for I2C2.
SDA2	I/O	ST	Synchronous serial data input/output for I2C2.
SOSCI	I	ST/CMOS	32.768 kHz low-power oscillator crystal input; CMOS otherwise.
SOSCO	O	—	32.768 kHz low-power oscillator crystal output.
TMS	I	ST	JTAG Test mode select pin.
TCK	I	ST	JTAG test clock input pin.
TDI	I	ST	JTAG test data input pin.
TDO	O	—	JTAG test data output pin.
T1CK	I	ST	Timer1 external clock input.
T2CK	I	ST	Timer2 external clock input.
T3CK	I	ST	Timer3 external clock input.
T4CK	I	ST	Timer4 external clock input.
T5CK	I	ST	Timer5 external clock input.
T6CK	I	ST	Timer6 external clock input.
T7CK	I	ST	Timer7 external clock input.
T8CK	I	ST	Timer8 external clock input.
T9CK	I	ST	Timer9 external clock input.
U1CTS	I	ST	UART1 clear to send.
U1RTS	O	—	UART1 ready to send.
U1RX	I	ST	UART1 receive.
U1TX	O	—	UART1 transmit.
U2CTS	I	ST	UART2 clear to send.
U2RTS	O	—	UART2 ready to send.
U2RX	I	ST	UART2 receive.
U2TX	O	—	UART2 transmit.
VDD	P	—	Positive supply for peripheral logic and I/O pins.
VCAP/VDDCORE	P	—	CPU logic filter capacitor connection.
VSS	P	—	Ground reference for logic and I/O pins.
VREF+	I	Analog	Analog voltage reference (high) input.
VREF-	I	Analog	Analog voltage reference (low) input.

Legend: CMOS = CMOS compatible input or output; Analog = Analog input; P = Power
ST = Schmitt Trigger input with CMOS levels; O = Output; I = Input

Figura 8.1.7

Encapsulado de este chip:



Units		MILLIMETERS		
Dimension Limits		MIN	NOM	MAX
Number of Leads	N		100	
Lead Pitch	e		0.50 BSC	
Overall Height	A	–	–	1.20
Molded Package Thickness	A2	0.95	1.00	1.05
Standoff	A1	0.05	–	0.15
Foot Length	L	0.45	0.60	0.75
Footprint	L1		1.00 REF	
Foot Angle	ϕ	0°	3.5°	7°
Overall Width	E		16.00 BSC	
Overall Length	D		16.00 BSC	
Molded Package Width	E1		14.00 BSC	
Molded Package Length	D1		14.00 BSC	
Lead Thickness	c	0.09	–	0.20
Lead Width	b	0.17	0.22	0.27
Mold Draft Angle Top	α	11°	12°	13°
Mold Draft Angle Bottom	β	11°	12°	13°

Figura 8.1.8

Luego, tras el estudio las seis páginas anteriores, es posible saber de qué dispone y de cómo se debe conectar el controlador.

Por lo que se puede pasar a la siguiente etapa del diseño: Profundizar más en detalle en los elementos externos que se van a conectar al sistema basado en este controlador para aclarar la necesidad de incluirlos en el diseño y por consiguiente, de su funcionamiento en la aplicación.

8.2. BUS CAN

Introducción:

CAN (acrónimo del inglés Controller Area Network) es un protocolo de comunicaciones desarrollado por la firma alemana Robert Bosch GmbH, basado en una topología bus para la transmisión de mensajes en entornos distribuidos, ofreciendo una solución a la gestión de la comunicación entre múltiples CPUs (unidades centrales de proceso).

CAN fue desarrollado, inicialmente para aplicaciones en los automóviles y por lo tanto la plataforma del protocolo es resultado de las necesidades existentes en el área de la automoción. La Organización Internacional para la Estandarización (ISO, International Organization for Standarization) define dos tipos de redes CAN: una red de alta velocidad (hasta 1 Mbps), bajo el estándar ISO 11898-2, destinada para controlar el motor e interconectar la unidades de control electrónico (ECU); y una red de baja velocidad tolerante a fallos (menor o igual a 125 Kbps), bajo el estándar ISO 11519-2/ISO 11898-3, dedicada a la comunicación de los dispositivos electrónicos internos de un automóvil como son control de puertas, techo corredizo, luces y asientos.

CAN es un protocolo de comunicaciones serie que soporta control distribuido en tiempo real con un alto nivel de seguridad y multiplexación.
El establecimiento de una red CAN para interconectar los dispositivos electrónicos internos de un vehículo tiene la finalidad de sustituir o eliminar el cableado. Las ECUs, sensores, sistemas antideslizantes, etc. se conectan mediante una red CAN a velocidades de transferencia de datos de hasta 1 Mbps.
De acuerdo al modelo de referencia OSI (Open Systems Interconnection, Modelo de interconexión de sistemas abiertos), la arquitectura de protocolos CAN incluye tres capas: física, de enlace de datos y aplicación, además de una capa especial para gestión y control del nodo llamada capa de supervisor.

Donde el protocolo de comunicaciones CAN proporciona los siguientes beneficios:

- El procesador anfitrión (Host) delega la carga de comunicaciones a un periférico inteligente, por lo tanto el procesador anfitrión dispone de mayor tiempo para ejecutar sus propias tareas.
- Es un protocolo de comunicaciones normalizado, con lo que se simplifica y economiza la tarea de comunicar subsistemas de diferentes fabricantes sobre una red común o bus.
- Al ser una red multiplexada, reduce considerablemente el cableado y elimina las conexiones punto a punto, excepto en los enganches.

Principales características:

CAN se basa en el modelo productor/consumidor, el cual es un concepto, o paradigma de comunicaciones de datos, que describe una relación entre un productor y uno o más consumidores.
CAN es un protocolo orientado a mensajes, es decir, la información que se va a intercambiar se descompone en mensajes, a los cuales se les asigna un identificador y se encapsulan en tramas para su transmisión. Cada mensaje tiene un identificador único dentro de la red, con el cual los nodos deciden aceptar o no dicho mensaje.
Dentro de sus principales características se encuentran:

- Prioridad de mensajes.
- Garantía de tiempos de latencia.
- Flexibilidad en la configuración.
- Recepción por multidifusión (multicast) con sincronización de tiempos.
- Sistema robusto en cuanto a consistencia de datos.
- Sistema multimaestro.
- Detección y señalización de errores.
- Retransmisión automática de tramas erróneas
- Distinción entre errores temporales y fallas permanentes de los nodos de la red, y desconexión autónoma de nodos defectuosos.

En la siguiente figura se observa el esquema de un bus CAN, donde se aprecian las dos líneas de bus, una para transmisión (TX) y la otra para recepción (RX), que se conectan a varios módulos con los que se podrán comunicar siguiendo un protocolo CSMA.

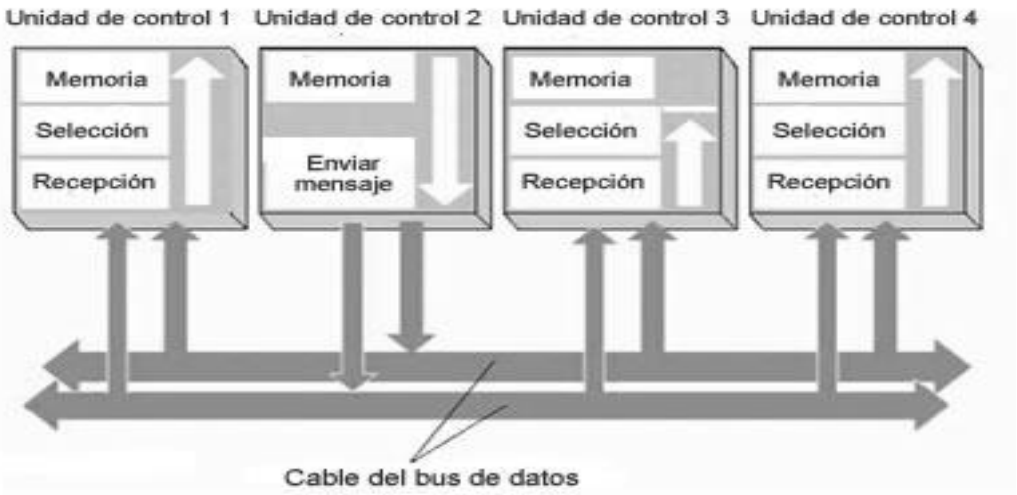


Figura 8.2.1

Debido a que dentro de un mismo vehículo pueda haber distintas necesidades por diferentes dispositivos, ya que el sistema de ABS debe de tener una reacción más rápida y prioritaria que el sistema de gestión de temperatura, a parte de un sistema de priorización dentro de las tramas de la línea CAN, puede haber diferenciación en las frecuencias a las que se gestionen los dispositivos dentro del vehículo, incluyendo varios módulos CAN e incluso varios controladores para garantizar la seguridad ante fallos. Esto se traduce en seguridad y confort para el usuario.

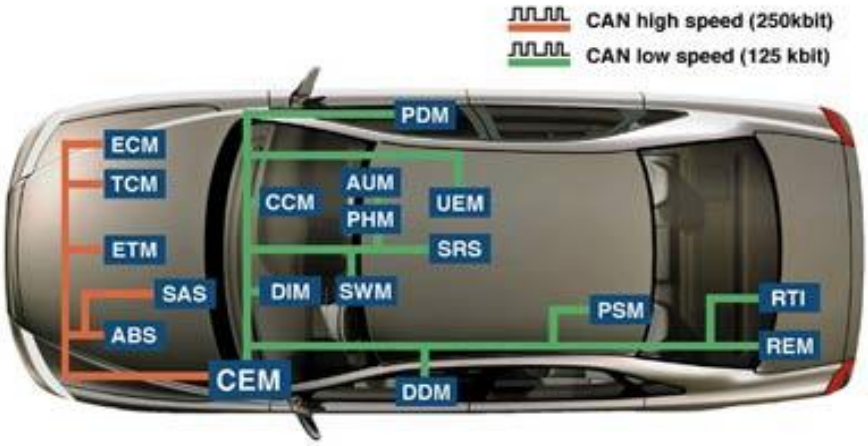


Figura 8.2.2

Capa física:

La capa física es responsable de la transferencia de bits entre los distintos módulos que componen la red. Define aspectos como niveles de señal, codificación, sincronización y tiempos en que los bits se transfieren al bus.

En la especificación original del CAN, la capa física no fue definida, permitiendo diferentes opciones para la elección del tipo de utilidad y niveles eléctricos de transmisión. Las características de las señales eléctricas en el bus fueron establecidas más tarde por el estándar ISO 11898.

La especificación CiA (CAN in Automation) complementó las definiciones respecto al medio físico y conectores. Los módulos conectados al bus interpretan dos niveles lógicos denominados:

Dominante: La tensión diferencial (CAN_H – CAN_L) es del orden de 2V con CAN_H = 3,5V y CAN_L = 1,5V (nominales).

Recesivo: La tensión diferencia (CAN_H – CAN_L) es del orden de 0V con CAN_H = CAN_L = 2,5V (nominales)

Otra de las partes a destacar de esta comunicación es la velocidad con la que se transmiten los mensajes a través de la red. Lo habitual es tener buses de corta longitud, par así lograr un funcionamiento óptimo. Pero si por el contrario tenemos redes de largas longitudes esto irá mermando nuestra velocidad de transmisión debido a los retardos en línea, impedancias, tolerancias de los osciladores, etc. Para atenuar estas carencias se colocan en los extremos del bus impedancias de carga para proporcionar una mayor estabilidad.

A continuación se muestran en la tabla los valores típicos de transmisión según la longitud de los buses:

Velocidad	Tiempo de Bit	Longitud Máxima
1 Mbps	1 µS	30 m
800 Kbps	1,25 µS	50 m
500 Kbps	2 µS	100 m
250 Kbps	4 µS	250 m
125 Kbps	8 µS	500 m
50 Kbps	20 µS	1000 m
20 Kbps	50 µS	2500 m
10 Kbps	100 µS	5000 m

El número máximo de módulos no está limitado por la especificación básica y depende de las características de los controladores CAN (SJA1000). Las especificaciones de buses de campo lo limitan a 32 y 64.

Capa de enlace de datos:

Una de las características que distingue a CAN con respecto a otras normas, es su técnica de acceso al medio denominada como CSMA/CD+CR (Carrier Sense, Multiple Access/Colission Detection + Collision Resolution), también conocida como CSMA/CD+AMP (Carrier Sense Multiple Access with Collision Detection and Arbitration Message Priority).

El acceso al medio mediante técnicas de acceso múltiple y detección de conflicto evolucionaron desde el método ALOHA inicial hasta su consolidación como método de acceso al medio de las redes Ethernet, con técnica CSMA/CD. Este método añade una característica adicional: la resolución en colisión. En la técnica CSMA/CD utilizada en redes Ethernet ante colisión de varias tramas, todas se pierden, pero CAN resuelve la colisión con la supervivencia de una de las tramas que chocan en el bus. Además, la trama superviviente es aquella a la que se ha identificado como de mayor prioridad.

La resolución de esta colisión se basa en aplicar una función lógica determinista a cada bit, que se resuelve con la prioridad del nivel definido como bit del tipo dominante.

Definido el bit dominante como el equivalente al valor lógico ‘0’ y bit recesivo al nivel lógico ‘1’ (se trata de una función AND de todos los bits transmitidos simultáneamente). Cada transmisor escucha continuamente el valor presente en el bus, y se retira cuando ese valor no coincide con el que dicho transmisor ha forzado. Mientras hay coincidencia la transmisión continua, donde finalmente el mensaje con el identificador de máxima prioridad sobrevive. Los demás módulos reintentarán la transmisión lo antes posible. Por lo tanto la prioridad queda así determinada por el campo Identificador (el cual forma parte del mensaje).

Se ha de tener en cuenta que la especificación CAN de Bosch no dice cómo se ha de traducir cada nivel de bit (dominante o recesivo) a variable física. Cuando se utilizan dos cables trenzados, que es lo que se especifica según ISO 11898, el nivel dominante es una tensión diferencial positiva en el bus, el nivel recesivo es ausencia de tensión o cierto valor negativo.

Referente al enlace de datos, se ve en la siguiente figura cómo evolucionó la topología CAN en las redes Ethernet de modelo OSI/ISO. Inicialmente había sólo tres capas: Aplicación, datos y física. Posteriormente surgieron las demás para adaptarse a las necesidades requeridas en las redes Ethernet, más complejas que las dadas en CAN.

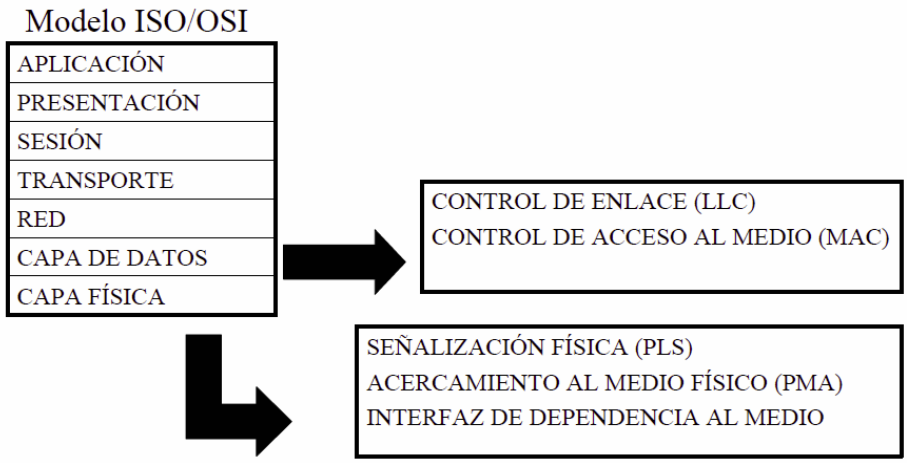


Figura 8.2.3

Capa de aplicación: Existen diferentes estándares que definen la capa de aplicación; algunos son muy específicos y están relacionados con sus campos de aplicación. Entre las capas de aplicación más utilizadas cabe mencionar CAL, CANopen, DeviceNet, SDS (Smart Distributed System), OSEK, CANKingdom.

Cuando un nodo necesita enviar información a través de una red CAN, puede ocurrir que varios nodos intenten transmitir simultáneamente. CAN resuelve lo anterior al asignar prioridades mediante el identificador de cada mensaje, donde dicha asignación se realiza durante el diseño del sistema en forma de números binarios y no puede modificarse dinámicamente. El identificador con el menor número binario es el que tiene mayor prioridad.

Mensajes y tipos de tramas:

El mensaje es una sucesión de unos y ceros representados por distintos niveles de tensión en los cables del bus. CAN utiliza mensajes de estructura predefinida para la gestión de la comunicación llamados Tramas.
Se distinguen entre dos variantes de CAN, el definido CAN 2.A ó CAN Standard (Basic CAN) y el definido CAN 2.B ó CAN Extendido (PeliCAN). Los formatos de trama son casi idénticos, con la diferencia del número de bits que se utilizan para el identificador de mensaje: 11 bits (2032 identificadores) en CAN Standard y 29 bits (536.870.912 identificadores) en CAN Extendido.
Las tramas CAN son de longitud reducida. La trama más larga es de 130 bits en CAN Standard y de 154 bits en CAN Extendido.

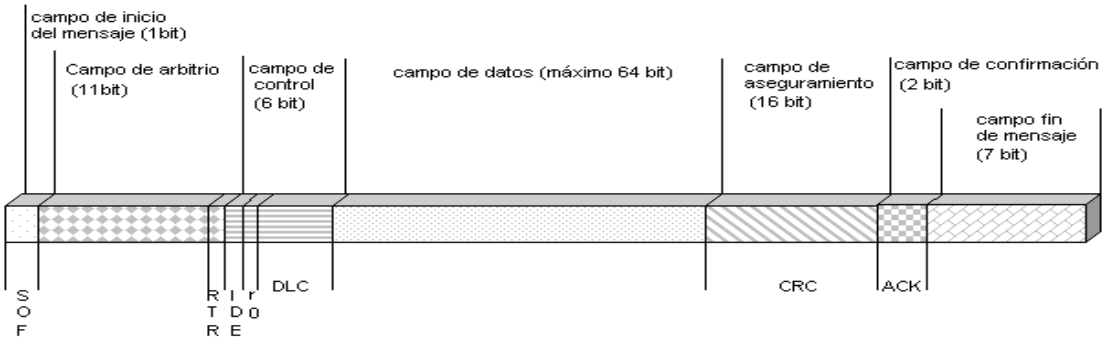


Figura 8.2.4

Los tipos de trama y estados del bus utilizados son:

- Trama de datos

Es aquella que un módulo utiliza para poner información en el bus. Puede incluir entre 0 y 8 bytes de información útil.

- Trama de interrogación remota

Puede ser utilizada por un módulo para solicitar la transmisión de una trama de datos con la información implicada a un identificador dado. El módulo que disponga de la información definida por el identificador la transmitirá en una trama de datos.

- Trama de error

Se usan para señalarle al resto de módulos que se ha detectado un error.

- Trama de sobrecarga

Permite que un módulo fuerce a los demás a alargar el tiempo entre transmisión de tramas sucesivas.

- Espaciado inter-tramas (IFS)

Las tramas de datos (y de interrogación remota) se separan entre sí por una secuencia predefinida que se denomina espaciado inter-trama.

- Bus en reposo

En los intervalos de inactividad se mantiene constantemente el nivel recesivo del bus.

En un bus CAN los módulos transmiten la información sin necesidad de una orden, simplemente mediante tramas de datos, bien sea por un proceso realizado cada cierto tiempo bien activado ante algún suceso en el módulo. La trama de interrogación remota sólo se suele utilizar para detectar la presencia de módulos o para actualizar la información en un módulo recién incorporado a la red. Los mensajes pueden entrar en colisión en el bus, pero como ya sabemos, el del identificador de mayor prioridad prevalecerá en ese instante, y los demás tendrán que retransmitir sus datos posteriormente.

Valor recesivo y valor dominante:

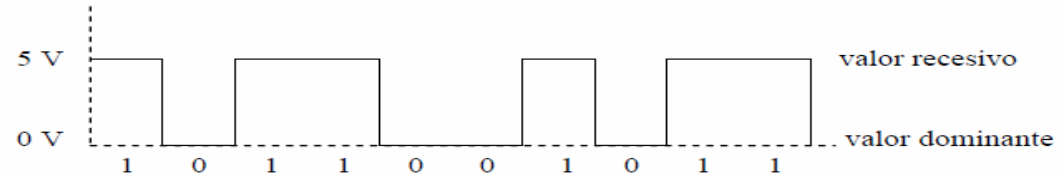


Figura 8.2.5

Formatos de Trama:

El mensaje se descompone en campos de diferente tamaño que nos permitirán llevar a cabo el proceso de comunicación entre los módulos según el protocolo predefinido para el bus CAN. Esto facilita identificar el módulo emisor, el inicio y el final de la trama del mensaje, los datos, etc.

Los mensajes son introducidos en la línea con una cadencia que oscila entre los 7 y los 20 milisegundos, dependiendo de la velocidad del módulo que los introduce.

Trama de datos: Estructura del mensaje

Campo de inicio de mensaje: El mensaje se inicia con un bit dominante cuyo flanco descendente es utilizado por los módulos para sincronizarse entre sí.

Campo de arbitrio: Los 11 bits de este campo se usan como identificador para que los módulos puedan conocer la prioridad del mensaje. Cuanto más bajo sea el valor de este identificador, más alta será la prioridad, lo cual determinará el orden en el que van a ser introducidos los mensajes en la línea.

El bit RTR indica si el mensaje contiene datos (RTR=0) ó si se trata de una trama remota sin datos (RTR=1). Una trama de datos siempre tiene una prioridad más alta que una remota. La trama remota se usa para pedir datos a otros módulos o bien porque se necesitan para realizar un chequeo.

Campo de control: Este campo informa sobre las características del campo de datos. El bit IDE indica que se trata de una trama estándar (BasicCAN, 11bits) cuando es un ‘0’ e indica que es una trama extendida (PeliCAN, 29 bits) cuando es un ‘1’. El segundo bit (RB0) es siempre recesivo. Los cuatro bits que componen el campo DLC indican el número de bytes contenido en el campo de datos (0 a 8).

Campo de datos: En este campo aparece la información del mensaje con los datos que el módulo correspondiente introduce en la línea CAN. Puede contener entre 0 a 8 bytes (de 0 a 64 bits).

Campo de aseguramiento (CRC): Es el código de redundancia cíclica que genera el transmisor por la división módulo 2 de todos los bits precedentes del mensaje, incluyendo los de relleno si existen por el polinomio generador: X15+ X14+ X8+ X7+ X4+ X3+ X1+1, el resto de esta división es el código CRC transmitido. Los receptores comprueban este código. Este campo tiene una longitud de 16 bits y se utiliza para detectar errores por los 15 primeros, mientras el último siempre será un bit recesivo (1) para delimitar el campo CRC.

Campo de confirmación (ACK): El campo ACK se compone de dos bits que son siempre transmitidos como recesivos (1). Todos los módulos que reciben el mismo CRC modifican el primer bit del campo ACK por uno dominante (0), de forma que el módulo que está todavía transmitiendo reconoce que al menos un módulo ha recibido un mensaje correctamente. De no ser así, el módulo transmisor interpreta que su mensaje tiene un error.

Campo de final de mensaje (EOF): Este campo indica el final del mensaje con una cadena de 7 bits recesivos. Nos puede suceder que en determinados mensajes se produzcan largas cadenas de ceros y unos, y que esto nos provoque una pérdida de sincronización entre los módulos. El protocolo CAN resuelve esta situación metiendo un bit de diferente valor cada cinco bits iguales. El módulo que utilice el mensaje, descarta un bit posterior a cinco bits iguales. Estos bits reciben el nombre de “stuffing”.

Espaciado entre tramas (IFS): Consta de un mínimo de 3 bits recesivos, es decir, lleva tres unos lógicos como mínimo antes de entregar la siguiente trama para permitir distinguirlas.

Trama remota: El formato es análogo a la trama de datos pero con el bit RTR recesivo. Por otra parte una trama remota no incluye nunca datos. El identificador es el del mensaje que se solicita, el campo longitud corresponde a la longitud de ese mensaje.

Trama de error: Las tramas de error son generadas por cualquier módulo que detecta un error. Consiste en dos campos: Indicador de error (Error Flag) y Delimitador de error. El delimitador de error consta de 8 bits recesivos consecutivos y permite a los módulos reiniciar la comunicación de nuevo tras el error. El indicador de error varía según el estado error:

Si un módulo en estado de error "activo" detecta un error en el bus interrumpe la comunicación del mensaje en proceso generando un "indicador de error activo" que consiste en una secuencia de 6 bits dominantes sucesivos. Esta secuencia rompe la regla de relleno de bits y provocará la generación de tramas de error en otros módulos. Por tanto, el indicador de error puede extenderse entre 6 y 12 bits dominantes sucesivos. Finalmente se espera el campo que delimita el error formado por lo 8 bits recesivos. Entonces la comunicación se reinita y el módulo que había sido interrumpido reintenta la transmisión del mensaje.

Si un módulo en estado de error "pasivo" detecta un error, el módulo transmite un "indicador de error pasivo" seguido de nuevo por el campo delimitador de error.
El indicador de error de tipo pasivo consiste en 6 bits recesivos seguidos, y por tanto, la trama de error para un módulo pasivo es una secuencia de 14 bits recesivos. De aquí se deduce que la transmisión de una trama de error de tipo pasivo no afectará a ningún módulo en la red, excepto cuando el error es detectado por el propio módulo que está transmitiendo. En ese caso los demás módulos detectarán una violación de las reglas de relleno y transmitirán a su vez tramas de error.

Tras señalar un error por medio de la trama de error apropiada, cada módulo transmite bits recesivos hasta que recibe un bit también recesivo, luego transmite 7 bits recesivos consecutivos antes de finalizar el tratamiento de error.

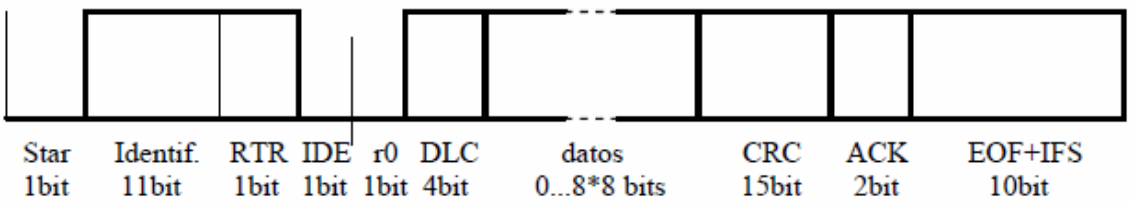


Figura 8.2.6

Espaciado entre tramas

El espaciado entre tramas separa una trama (de cualquier tipo) de la siguiente trama de datos o trama remota. El espaciado entre tramas ha de constar de al menos 3 bits recesivos. Esta secuencia de bits se denomina "inter-trama" (IFS). Una vez transcurrida esta secuencia, un módulo en estado de error activo puede iniciar una nueva transmisión o el bus permanecerá en reposo. Para un módulo en estado de error pasivo la situación es diferente, deberá esperar una secuencia adicional de 8 bits recesivos antes de poder iniciar una transmisión. De esta forma, se asegura una ventaja en inicio de transmisión a los módulos en estado activo frente a los módulos en estado pasivo.

Trama de sobrecarga

Una trama de sobrecarga tiene el mismo formato que una trama de error activo. Sin embargo, la trama de sobrecarga sólo puede generarse durante el espacio entre tramas. De esta forma se diferencia de una trama de error, que sólo puede ser transmitida durante la transmisión de un mensaje. La trama de sobrecarga consta de dos campos, el Indicador de sobrecarga y del Delimitador.
El Indicador de sobrecarga consta de 6 bits dominantes que pueden ser seguidos por los generados por otros módulos, dando lugar a un máximo de 12 bits dominantes.
El Delimitador es de 8 bits recesivos.

Una trama de sobrecarga puede ser generada por cualquier módulo que debido a sus condiciones internas no está en condiciones de hincar la recepción de un mensaje.
De esta forma retrasa el inicio de la transmisión de un nuevo mensaje. Un módulo puede generar como máximo 2 tramas de sobrecarga consecutivas para retrasar un mensaje. Otra razón para iniciar la transmisión de una trama de sobrecarga es la detección por cualquier módulo de un bit dominante en los 3 bits de "inter-trama". Por todo ello una trama de sobrecarga generada por un módulo dará normalmente lugar a la generación de tramas de sobrecarga por los demás módulos dando posteriormente lugar a, como se ha indicado, un máximo de 12 bits dominantes de indicador de sobrecarga.

Arbitraje

Un módulo transmisor monitoriza constantemente el estado del bus. Durante la transmisión del campo Arbitraje, la detección de un bit dominante cuando el bit transmitido ha sido recesivo, hace que el módulo detenga la transmisión y pase a recepción de la trama. Así no se pierde información y no se destruye por colisión ninguna trama de datos o de tipo remota.

En un bus único, un identificador de mensaje ha de tener asignado a un solo módulo concreto, es decir, ha de evitar que dos módulos puedan iniciar la transmisión simultánea de mensajes con el mismo identificador y datos diferentes. La filosofía CAN es de qué un mensaje es único en el sistema. Las tramas remotas con identificador concreto que puedan ser generadas por cualquier módulo han de coincidir en cuanto al campo longitud, definiendo un mensaje como el conjunto:

identificador + longitud del campo de datos + contenido de los datos

El mensaje ha de ser único en el sistema y además estar asignado a un módulo concreto. CAN establece dos formatos de tramas de datos (data frame) que difieren en la longitud del campo del identificador, las tramas estándares (standard frame) con un identificador de 11 bits definidas en la especificación CAN 2.0A, y las tramas extendidas (extended frame) con un identificador de 29 bits definidas en la especificación CAN 2.0B.
Para la transmisión y control de mensajes CAN, se definen cuatro tipos de tramas: de datos, remota (remote frame), de error (error frame) y de sobrecarga (overload frame). Las tramas remotas también se establecen en ambos formatos, estándar y extendido, y tanto las tramas de datos como las remotas se separan de tramas precedentes mediante espacios entre tramas (interframe space).
En cuanto a la detección y manejo de errores, un controlador CAN cuenta con la capacidad de detectar y manejar los errores que surjan en una red. Todo error detectado por un nodo, se notifica inmediatamente al resto de los nodos.

Capa de supervisor: La sustitución del cableado convencional por un sistema de bus serie presenta el problema de que un nodo defectuoso puede bloquear el funcionamiento del sistema completo. Cada nodo activo transmite una bandera de error cuando detecta algún tipo de error y puede ocasionar que un nodo defectuoso pueda acaparar el medio físico. Para eliminar este riesgo el protocolo CAN define un mecanismo autónomo para detectar y desconectar un nodo defectuoso del bus, dicho mecanismo se conoce como aislamiento de fallos.

8.3. BUS I²C

Introducción:

I²C es un bus de comunicaciones en serie. Su nombre viene de Inter-Integrated Circuit (Circuitos Inter-Integrados). La versión 1.0 data del año 1992 y la versión 2.1 del año 2000, su diseñador es Philips. La velocidad es de 100Kbits por segundo en el modo estándar, aunque también permite velocidades de hasta 3.4 Mbit/s. Es un bus muy usado en la industria, principalmente para comunicar microcontroladores y sus periféricos en sistemas integrados (Embedded Systems) y generalizando más para comunicar circuitos integrados entre si que normalmente residen en un mismo circuito impreso.
La principal característica de I²C es que utiliza dos líneas para transmitir la información: una para los datos y por otra la señal de reloj. También es necesaria una tercera línea, pero esta sólo es la referencia (masa). Como suelen comunicarse circuitos en una misma placa que comparten una misma masa esta tercera línea no suele ser necesaria.

Incluye:

- Una línea de reloj (SCL –System Clock)
- Una línea de datos (SDA – System Data)
- Una línea de masa (GND – Earth)

Las líneas SCL y SDA son del tipo drenador abierto, es decir, un estado similar al de colector abierto, pero asociadas a un transistor de efecto de campo (FET). Se deben polarizar en estado alto (conectándolas a la alimentación por medio de resistores de *pull-up*) lo que define una estructura de bus que permite conectar en paralelo múltiples entradas y salidas.

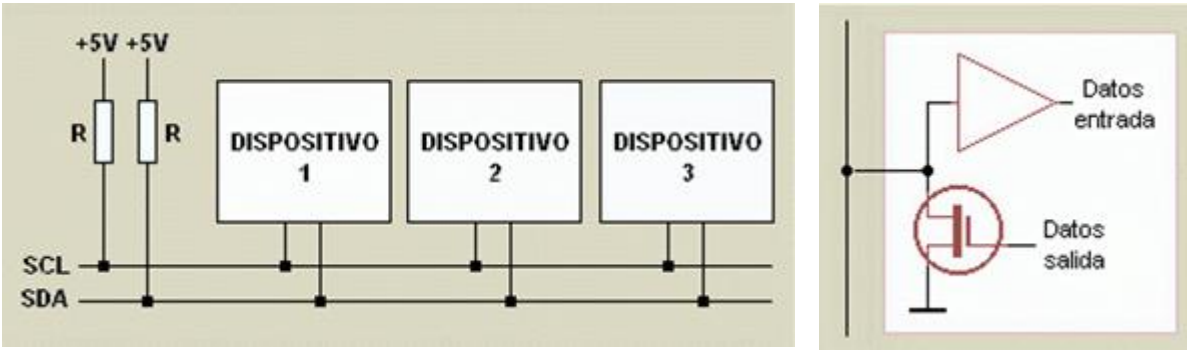


Figura 8.3.1

Las dos líneas del bus están a nivel alto cuando están inactivas. En principio, el número de dispositivos que se pueden conectar al bus no tiene límites, aunque hay que observar que la capacidad máxima sumada de todos los dispositivos no supere los 400pF. El valor de los resistores de polarización no es muy crítico, y puede ir desde 1K8 (1.800 ohms) a 47K (47.000 ohms). Un valor menor de resistencia incrementa el consumo de los integrados pero disminuye la sensibilidad al ruido y mejora el tiempo de los flancos de subida y bajada de las señales. Los valores más comunes en uso son entre 1K8 y 10K.

Protocolo de comunicación del bus I2C:

Habiendo varios dispositivos conectados en el bus, es lógico que para establecer una comunicación a través de él se deba respetar un protocolo. Es importante saber que existen dispositivos maestros y dispositivos esclavos, pero sólo los dispositivos maestros pueden iniciar una comunicación.

La condición inicial, de bus libre, es cuando ambas señales están en estado lógico alto. En este estado cualquier dispositivo maestro puede ocuparlo, estableciendo la condición de inicio (start). Esta condición se presenta cuando un dispositivo maestro pone en estado bajo la línea de datos (SDA), pero dejando en alto la línea de reloj (SCL).

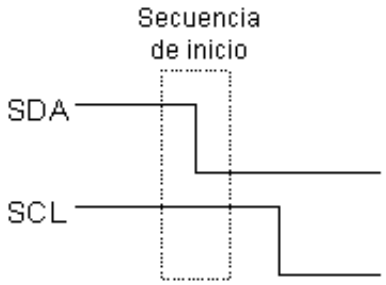


Figura 8.3.2

El primer byte que se transmite tras la condición de inicio contiene siete bits que componen la dirección del dispositivo que se desea seleccionar, y un octavo bit que corresponde a la operación que se quiere realizar con él (lectura o escritura).

Si el dispositivo cuya dirección corresponde a la que se indica en los siete bits (A0-A6) está presente en el bus, éste contesta con un bit a nivel bajo, situándolo inmediatamente después del octavo bit que ha enviado el dispositivo maestro. Este bit de reconocimiento (ACK) a nivel bajo le indica al dispositivo maestro que el esclavo reconoce la solicitud y está en condiciones de comunicarse. En este punto se establece la comunicación y comienza el intercambio de información entre los dispositivos.

En la figura 8.3.3 se puede ver la secuencia en la que se indica la dirección (address) del dispositivo con el que se desea comunicar el dispositivo maestro.

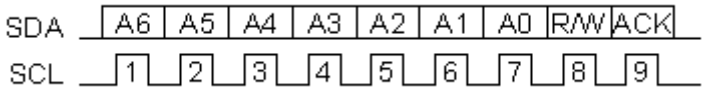


Figura 8.3.3

Escritura en un dispositivo esclavo:

Si el bit de lectura/escritura (R/W) fue puesto en esta comunicación a nivel lógico bajo (escritura), el dispositivo maestro envía datos al dispositivo esclavo. Esto se mantiene mientras continúe recibiendo señales de reconocimiento, y el contacto concluye cuando se hayan transmitido todos los datos.

Lectura de un dispositivo esclavo:

En el caso contrario, cuando el bit de lectura/escritura estaba a nivel lógico alto (lectura), el dispositivo maestro genera pulsos de reloj para que el dispositivo esclavo pueda enviar los datos. Luego de cada byte recibido el dispositivo maestro (quien está recibiendo los datos) genera un pulso de reconocimiento.

El dispositivo maestro puede dejar libre el bus generando una condición de parada (stop).

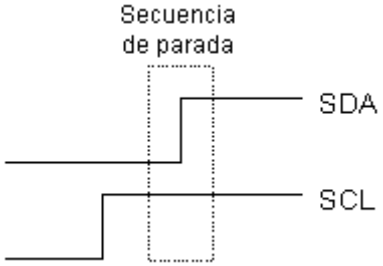


Figura 8.3.4

Si se desea seguir transmitiendo, el dispositivo maestro puede generar otra condición de inicio en lugar de una condición de parada. Esta nueva condición de inicio se denomina "inicio reiterado" y se puede emplear para direccionar un dispositivo esclavo diferente o para alterar el estado del bit de lectura/escritura.

Lo más común en los dispositivos para el bus I2C es que utilicen direcciones de 7 bits, aunque existen dispositivos de 10 bits (este último caso es raro).
Debe tenerse en cuenta que una dirección de 7 bits implica que se pueden poner hasta 128 dispositivos sobre un bus I2C, ya que un número de 7 bits puede ir desde 0 a 127.
Cuando se envían las direcciones de 7 bit, de cualquier modo la transmisión es de 8 bits. El bit extra se utiliza para informarle al dispositivo esclavo si el dispositivo maestro va a escribir o va a leer datos desde él. Si el bit de lectura/escritura (R/W) es cero, el dispositivo maestro está escribiendo en el esclavo. Si el bit es 1 el maestro está leyendo desde el esclavo. La dirección de 7 bit se coloca en los 7 bist más significativos del byte y el bit de lectura/escritura es el bit menos significativo.
El hecho de colocar la dirección de 7 bits en los 7 bits más significativos del byte produce confusiones entre quienes comienzan a trabajar con este bus. Si, por ejemplo, se desea escribir en la dirección 21 (hexadecimal), en realidad se debe enviar un 42, que es un 21 desplazado un bit hacia la izquierda. También se pueden tomar las direcciones del bus I2C como direcciones de 8 bit, en las que las pares son de sólo escritura y las impares son de sólo lectura.

Queremos escribir un 21h: 0 0 1 0 0 0 0 1

Debemos escribir un 42h: 0 1 0 0 0 0 1 0 → ya que el último bit es W/R

Las transacciones en el bus I^2C tienen por tanto este formato:

| Start | A7 A6 A5 A4 A3 A2 A1 | R/W | ACK | ... DATA ... | ACK | stop | idle |

Donde los pasos que sigue una comunicación mediante I^2C son:

1. El bus esta libre cuando SDA y SCL están en estado lógico alto.
2. En estado bus libre, cualquier dispositivo puede ocupar el bus I²C como maestro.
3. El maestro comienza la comunicación enviando un patrón llamado "start condition". Esto alerta a los dispositivos esclavos, poniéndolos a la espera de una transacción.
4. El maestro se dirige al dispositivo con el que quiere hablar, enviando un byte que contiene los siete bits (A7-A1) que componen la dirección del dispositivo esclavo con el que se quiere comunicar, y el octavo bit (A0) de menor peso se corresponde con la operación deseada (L/E), lectura=1 (recibir del esclavo) y escritura=0 (enviar al esclavo).
5. La dirección enviada es comparada por cada esclavo del bus con su propia dirección, si ambas coinciden, el esclavo se considera direccionado como esclavo-transmisor o esclavo-receptor dependiendo del bit R/W.
6. El esclavo responde enviando un bit de ACK que le indica al dispositivo maestro que el esclavo reconoce la solicitud y está en condiciones de comunicarse.
7. Seguidamente comienza el intercambio de información entre los dispositivos.
8. El maestro envía la dirección del registro interno del dispositivo que se desea leer o escribir.
9. El esclavo responde con otro bit de ACK
10. Ahora el maestro puede empezar a leer o escribir bytes de datos. Todos los bytes de datos deben constar de 8 bits, el número máximo de bytes que pueden ser enviados en una transmisión no está restringido, siendo el esclavo quien fija esta cantidad de acuerdo a sus características.

11. Cada byte leído/escrito por el maestro debe ser obligatoriamente reconocido por un bit de ACK por el dispositivo maestro/esclavo.
12. Se repiten los 2 pasos anteriores hasta finalizar la comunicación entre maestro y esclavo.
13. Aun cuando el maestro siempre controla el estado de la línea del reloj, un esclavo de baja velocidad o que deba detener la transferencia de datos mientras efectúa otra función, puede forzar la línea SCL a nivel bajo. Esto hace que el maestro entre en un estado de espera, durante el cual, no transmite información esperando a que el esclavo esté listo para continuar la transferencia en el punto donde había sido detenida.
14. Cuando la comunicación finaliza, el maestro transmite una "stop condition" para dejar libre el bus.
15. Después de la "stop condition", es obligatorio para el bus estar idle durante unos microsegundos.

Ejemplo de lectura desde un dispositivo esclavo MLX90614:

Antes de leer datos desde el dispositivo esclavo, primero se le debe informar desde cuál de sus direcciones internas se va a leer. De manera que una lectura desde un dispositivo esclavo en realidad comienza con una operación de escritura en él. Es igual a cuando se desea escribir en él: Se envía la secuencia de inicio, la dirección de dispositivo con el bit de lectura/escritura en bajo y el registro interno desde el que se desea leer. Ahora se envía otra secuencia de inicio nuevamente con la dirección de dispositivo, pero esta vez con el bit de lectura/escritura en alto. Luego se leen todos los bytes necesarios y se termina la transacción con una secuencia de parada.

En el ejemplo del módulo del sensor de temperatura IR MLX90614, el registro de temperatura de su memoria RAM se debe leer de la siguiente forma:

- Enviar una secuencia de inicio.
- Enviar 0x00 (La dirección genérica del dispositivo con el bit de lectura/escritura en bajo).
- Enviar 0x07 (dirección interna del registro de temperatura 0-255).
- Enviar una secuencia de inicio (inicio reiterado).
- Enviar 0x01 (La dirección de dispositivo del MLX90614 con el bit de lectura/escritura en alto para poder leer).
- Leer los datos necesarios.
- Finalmente se debe enviar la secuencia de parada.

De manera gráfica, estas secuencias quedan así:

Figura 8.3.5

O bien, y de forma más detallada, puede recurrirse al catálogo del MLX90614 para ver la escritura en la EEPROM y lectura en la RAM:

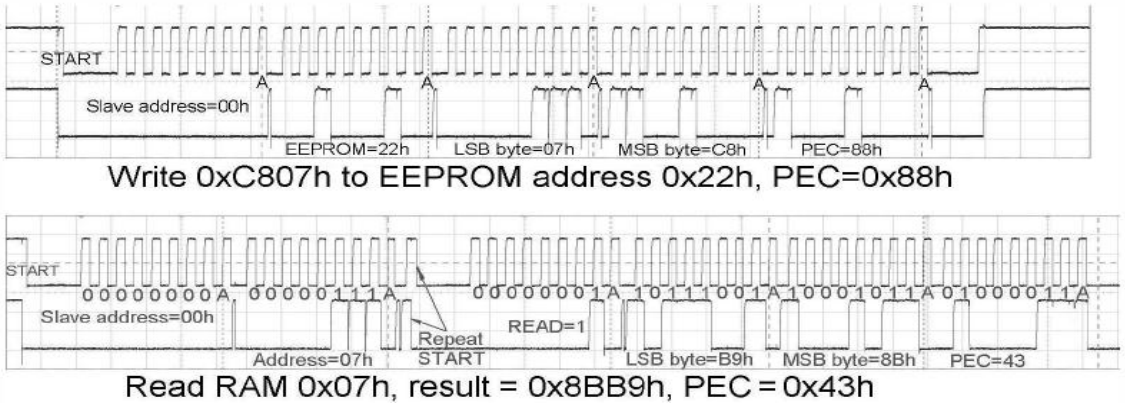


Figura 8.3.6

NOTA: Véase Pág. 16 del *datasheet* del sensor

8.4. BUS SPI

Introducción:

El Bus SPI (Serial Peripheral Interface) es un estándar de comunicaciones, usado principalmente para la transferencia de información entre circuitos integrados en equipos electrónicos. El bus de interfaz de periféricos serie o bus SPI es un estándar para controlar casi cualquier dispositivo electrónico digital que acepte un flujo de bits serie regulado por un reloj, pudiendo alcanzar una frecuencia de hasta 20MHz.

Incluye:

- Una línea de reloj (CLK)
- Dato entrante (SDI)
- Dato saliente (SDO)
- Pin de chip select (CS)

El Chip Select conecta o desconecta la operación del dispositivo con el que uno desea comunicarse. De esta forma, este estándar permite multiplexar las líneas de reloj. Muchos sistemas digitales tienen periféricos que necesitan existir pero no ser rápidos. La ventajas de un bus serie es que minimiza el número de conductores, pines y el tamaño del circuito integrado. Esto reduce el coste de fabricar montar y probar la electrónica. Un bus de periféricos serie es la opción más flexible cuando se tiene tipos diferentes de periféricos serie. El hardware consiste en señales de reloj, data in, data out y chip select para cada circuito integrado que tiene que ser controlado. Casi cualquier dispositivo digital puede ser controlado con esta combinación de señales. Los dispositivos se diferencian en un número predecible de formas. Unos leen el dato cuando el reloj sube otros cuando el reloj baja. Algunos lo leen en el flanco de subida del reloj y otros en el flanco de bajada. Escribir es casi siempre en la dirección opuesta de la dirección de movimiento del reloj. Algunos dispositivos tienen dos relojes. Uno para capturar o mostrar los datos y el otro para el dispositivo interno.

En las figuras mostradas a continuación se aprecia un módulo SPI interconectando módulos con un solo esclavo o con tres esclavos respectivamente.
Donde, como se puede diferenciar, un multiesclavo requiere un ChipSelect (CS) por cada módulo que sea añadido, manteniéndose las líneas de datos y de reloj comunes.

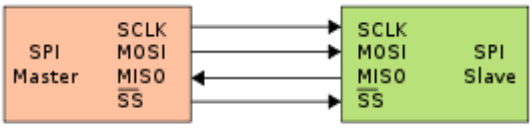


Figura 8.4.1

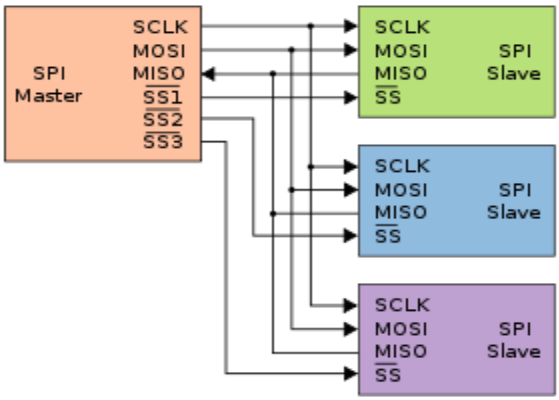


Figura 8.4.2

Dado que se pueden apreciar semejanzas entre el bus SPI y el I^2C, las ventajas y desventajas que puede aportar el bus SPI se muestran detalladas a continuación:

Ventajas del bus SPI:

- Comunicación Full Duplex.
- Mayor velocidad de transmisión que con I^2C o SMBus.
- Protocolo flexible en que se puede tener un control absoluto sobre los bits transmitidos.
- No está limitado a la transferencia de bloques de 8 bits.
- Elección del tamaño de la trama de bits, de su significado y propósito.
- Su implementación en hardware es extremadamente simple.
- Consume menos energía que I^2C o que SMBus debido que posee menos circuitos (incluyendo las resistencias pull-up) y estos son más simples.
- No es necesario arbitraje o mecanismo de respuesta ante fallos.
- Los dispositivos clientes usan el reloj que envía el servidor, no necesitan por tanto su propio reloj.
- No es obligatorio implementar un transceptor (emisor y receptor), un dispositivo conectado puede configurarse para que solo envíe, sólo reciba o ambas cosas a la vez.

- Usa mucho menos terminales en cada chip/conector que un interfaz paralelo equivalente.
- Como mucho una única señal específica para cada cliente (señal SS), las demás señales pueden ser compartidas.

Desventajas del bus SPI:

- Consume más pines de cada chip que I²C, incluso en la variante de 3 hilos.
- El direccionamiento se hace mediante líneas específicas (señalización fuera de banda) a diferencia de lo que ocurre en I²C que se selecciona cada chip mediante una dirección de 7 bits que se envía por las mismas líneas del bus.
- No hay control de flujo por hardware.
- No hay señal de asentimiento. El servidor podría estar enviando información sin que estuviese conectado ningún cliente y no se daría cuenta de nada.
- No permite fácilmente tener varios servidores conectados al bus.
- Sólo funciona en las distancias cortas a diferencia de, por ejemplo, RS-232, RS-485, o Bus CAN.

8.5. TIMERS

Los Timers (temporizadores) son elementos lógicos de frecuencia programable dedicados a medir el tiempo. Cuando transcurre el tiempo configurado hacen saltar algún mecanismo o función a modo de advertencia.
La necesidad de incluir estos elementos en cualquier sistema digital, facilita la funcionalidad, la precisión, así cómo la medición en cualquier aplicación.

Respecto a los Timers del MCU, se debe diferenciar entre el Timer1 y los demás, los cuales están semiemparejados en la arquitectura de este PIC.

Timer1

Es un temporizador de 16-bits, el cual puede servir como un contador de tiempo para un reloj en tiempo real ó bien puede operar como un contador/temporizador de intervalos.

Distingue tres modos de operación:

- Temporizador de 16 bits
- Contador síncrono de 16 bits
- Contador asíncrono de 16 bits

En la siguiente figura se muestra el diagrama de bloques del Timer1:

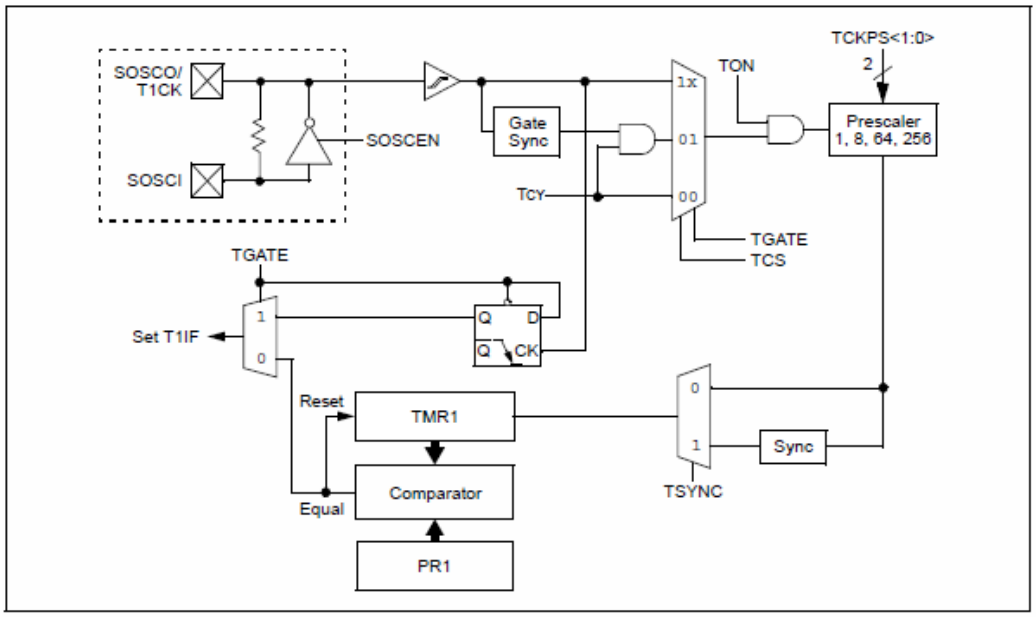


Figura 8.5.1

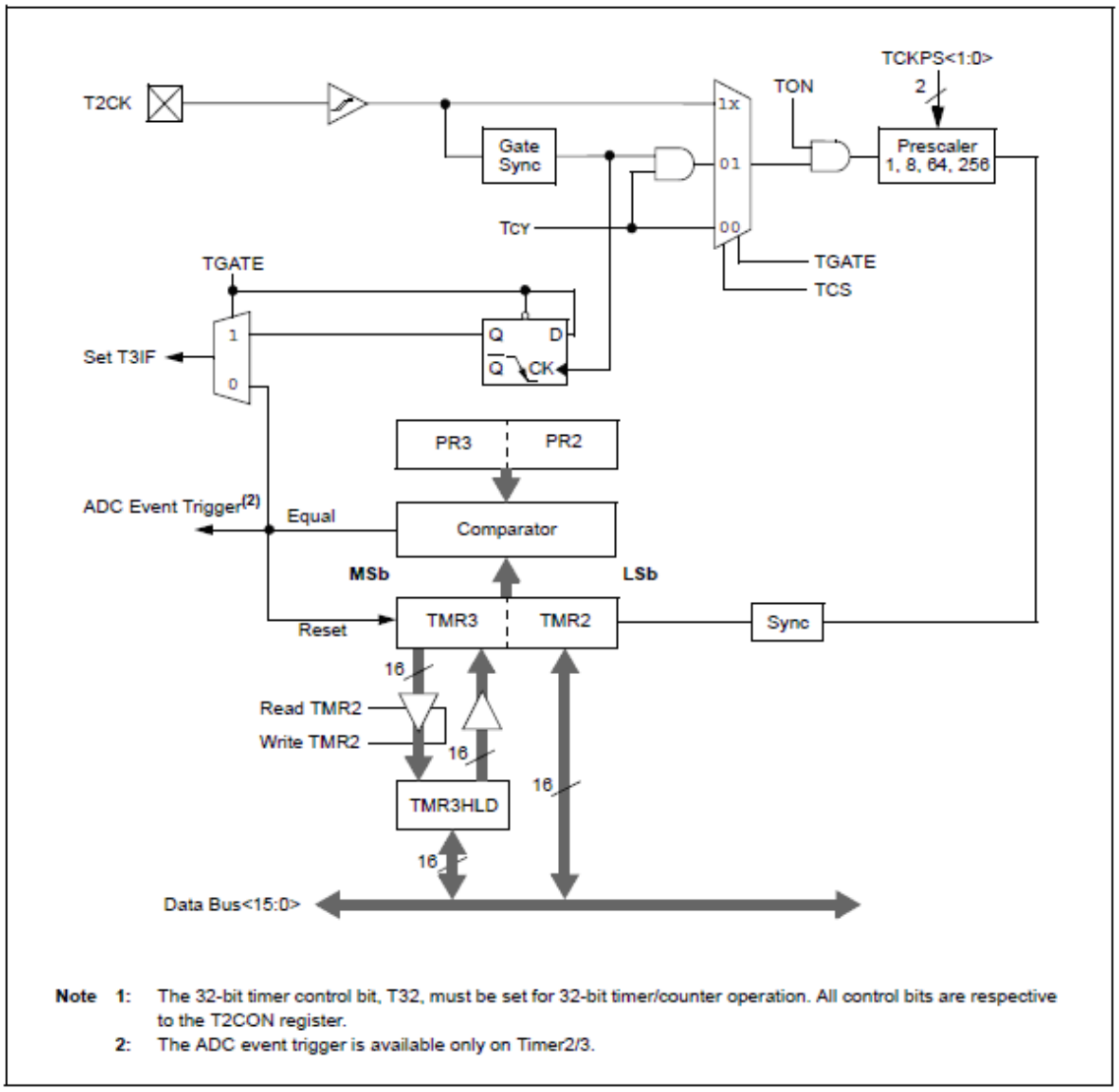
Timers 2/3, 4/5, 6/7, 8/9:

Son módulos de 32 bits que pueden ser configurados como temporizadores de 16 bits con un modo independiente de operación.

Como 32-bit timers distinguen tres modos de operación:

- 2 temporizadores independientes de 16 bits (excepto en modo asíncrono)
- Contador síncrono de 32 bits
- Contador asíncrono de 32 bits

En la siguiente figura se muestra el diagrama de bloques de estos Timers:



Note 1: The 32-bit timer control bit, T32, must be set for 32-bit timer/counter operation. All control bits are respective to the T2CON register.
2: The ADC event trigger is available only on Timer2/3.

Figura 8.5.2

8.6. INPUT CAPTURE

El módulo de captura de entrada se emplea para aplicaciones que requieren medidas de frecuencias o de simplemente pulsos acaecidos en la entrada.
Este módulo captura el valor en 16 bits cuando sucede un evento en el pin ICx.

Dichas capturas distinguen tres modos:

1. Modo de captura de eventos simple.
   1.1. Captura el valor del timer en el flanco de bajada dado en ICx.
   1.2. Captura el valor del timer en el flanco de subida dado en el ICx.

2. Modo de captura del valor del timer en cada flanco (subida y bajada).
3. Modo de captura de eventos con preescaler.
   3.1. Captura el valor del timer en cada cuarto flanco de subida dado en ICx.
   3.2. Captura el valor del timer en cada dieciseisavo flanco de subida en ICx.

Cada canal de captura de entrada puede elegir para la base de tiempos uno de estos dos timers de 16 bits: Timer2 o Timer3. Además, cada timer seleccionado puede elegir entre un reloj interno o un reloj externo.

En la siguiente figura se muestra el diagrama de bloques del módulo de captura:

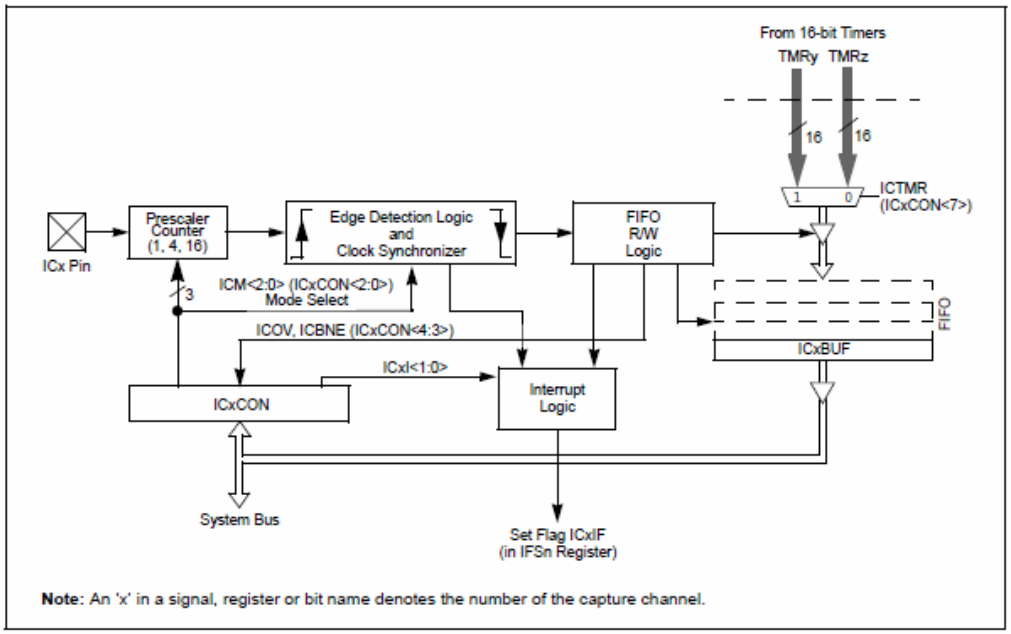


Figura 8.6.1

Aunque no se están empleando directamente para mediciones estos módulos de captura, se podrían utilizar para medir la temperatura de los sensores, ya que los propios sensores añaden un modo de configuración donde continuamente sacarían la temperatura que están leyendo mediante un pulso PWM, es decir, mediante un frecuencia que se podría medir con este modo e internamente calcular la relación.

8.7. ADCs

La conversión analógica-digital (CAD) consiste en la transcripción de señales analógicas en señales digitales, con el propósito de facilitar su procesamiento (codificación, compresión, etc.) y hacer la señal resultante (la digital) más inmune al ruido y otras interferencias a las que son más sensibles las señales analógicas.

La digitalización o conversión analógica-digital (conversión A/D) consiste básicamente en realizar de forma periódica medidas de la amplitud (tensión) de una señal (por ejemplo, la que proviene de un micrófono si se trata de registrar sonidos, de un sismógrafo si se trata de registrar vibraciones o de una sonda de un osciloscopio para cualquier nivel variable de tensión de interés), redondear sus valores a un conjunto finito de niveles preestablecidos de tensión (conocidos como niveles de cuantificación) y registrarlos como números enteros en cualquier tipo de memoria o soporte. La conversión A/D también es conocida por el acrónimo inglés ADC (analogue to digital converter).

En esta definición están patentes los cuatro procesos que intervienen en la conversión analógica-digital:

Muestreo (sampling): Consiste en tomar muestras periódicas de la amplitud de onda. La velocidad con que se toma esta muestra, es decir, el número de muestras por segundo, es lo que se conoce como frecuencia de muestreo.

Retención (hold): Las muestras tomadas han de ser retenidas (retención) por un circuito de retención (hold), el tiempo suficiente para permitir evaluar su nivel (cuantificación). Desde el punto de vista matemático este proceso no se contempla, ya que se trata de un recurso técnico debido a limitaciones prácticas, y carece, por tanto, de modelo matemático.

Cuantificación (quantification): en el proceso de cuantificación se mide el nivel de voltaje de cada una de las muestras. Consiste en asignar un margen de valor de una señal analizada a un único nivel de salida. Incluso en su versión ideal, añade, como resultado, una señal indeseada a la señal de entrada: el ruido de cuantificación.

Codificación (encoding): La codificación consiste en traducir los valores obtenidos durante la cuantificación al código binario. Hay que tener presente que el código binario es el más utilizado, pero también existen otros tipos de códigos que también son utilizados.

Este proceso de digitalización sigue el desarrollo planteado en la siguiente figura:

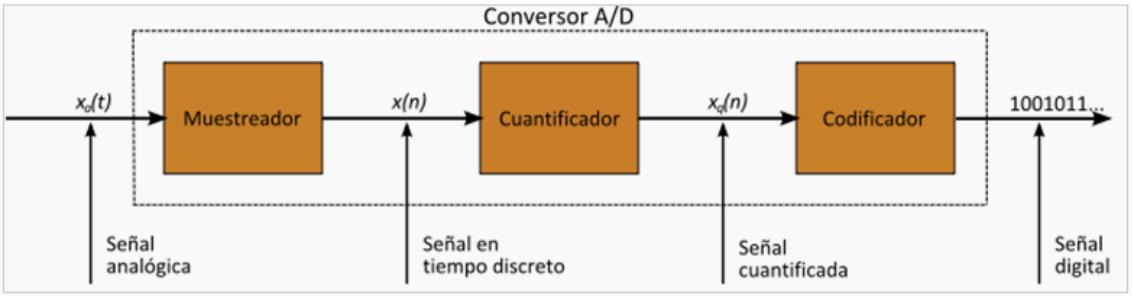


Figura 8.7.1

El MCU elegido dispone de hasta 32 entradas de ADC, separadas en dos módulos que multiplexan las entradas, donde cada conjunto dispone de sus propios SFR (Special Function Register).

El bit AD12B del registro ADxCON1<10> permite que cada módulo sea configurado por el usuario como:

- 10-bit, 4 sample/hold ADC (configuración por defecto)
- 12-bit, 1 sample/hold ADC

Donde para modificar dicho bit AD12B, es necesario que el módulo ADC esté deshabilitado

En la siguiente figura se muestra el diagrama de bloques de los ADCs:

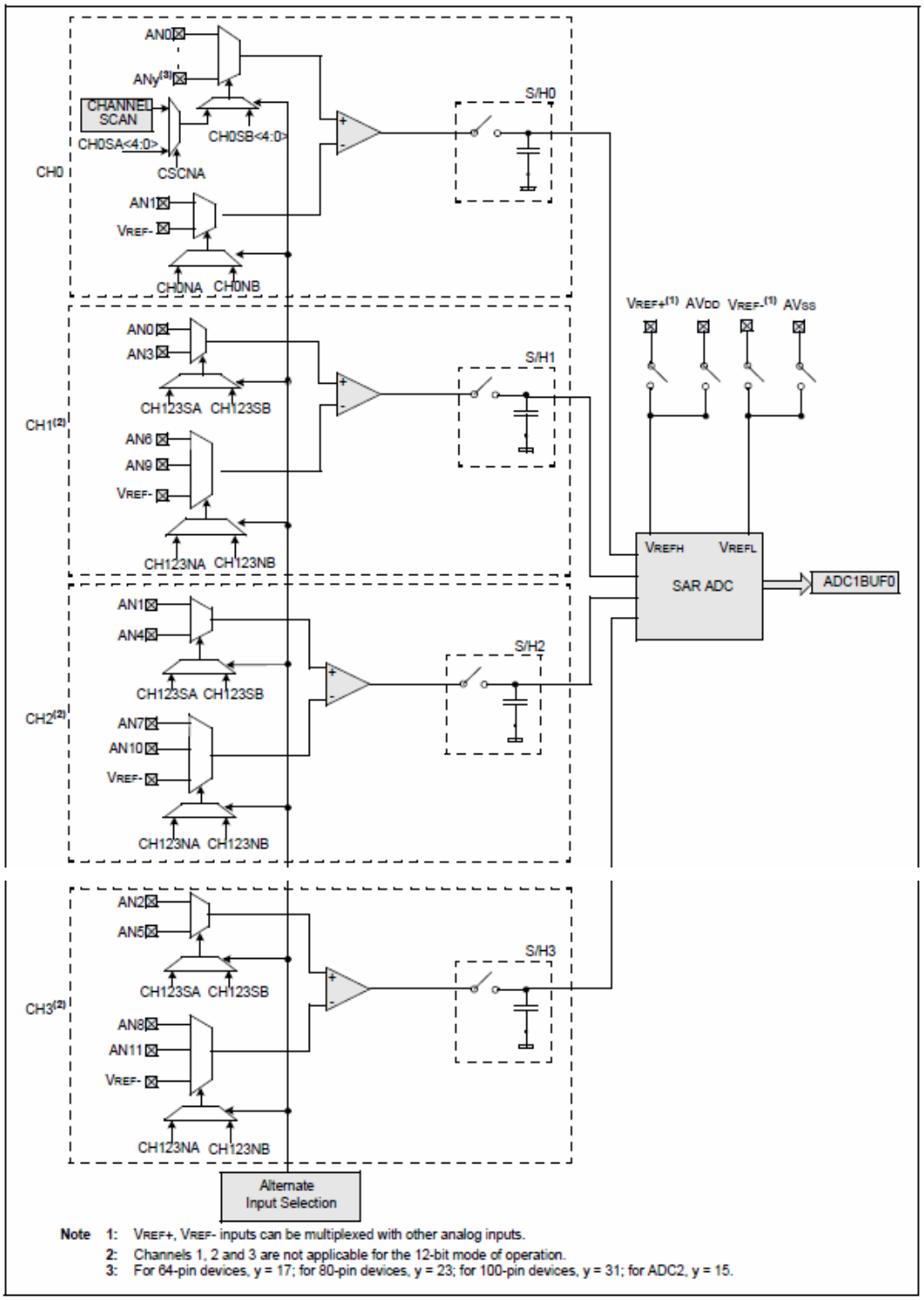


Figura 8.7.2

8.8. OSCILADOR

Un oscilador es un sistema capaz de crear perturbaciones o cambios periódicos o cuasi periódicos en un medio.
En electrónica un oscilador es un circuito que es capaz de convertir la corriente continua en una corriente que varía de forma periódica en el tiempo (corriente periódica); estas oscilaciones pueden ser senoidales, cuadradas, triangulares, etc., dependiendo de la forma que tenga la onda producida. Un oscilador de onda cuadrada suele denominarse multivibrador y por lo tanto, se les llama osciladores sólo a los que funcionan en base al principio de oscilación natural que constituyen una bobina L (inductancia) y un condensador C (capacitancia), mientras que a los demás se le asignan nombres especiales.

El oscilador se encarga de introducir una frecuencia de referencia en nuestro MCU que el propio MCU puede gestionar a través de multiplicadores y divisores. Esta frecuencia se puede introducir mediante un oscilador cerámico y mediante un cristal oscilador.

Además hay que tener en cuenta que el consumo del controlador PIC, y por tanto la cantidad de calor que disipe el componente depende del oscilador primario, donde:

Frecuencia de 4MHz → Consumo de 11mA → Low power
Frecuencia de 72MHz → Consumo de 64mA → High performance

Se puede alcanzar un alto rendimiento con un oscilador incluso de sólo 8MHz:

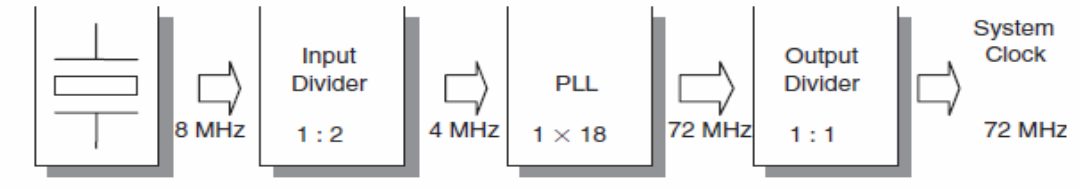


Figura 8.8.1

En la siguiente figura se muestra el diagrama de bloques de gestión del oscilador primario dentro de los dsPIC33FJ:

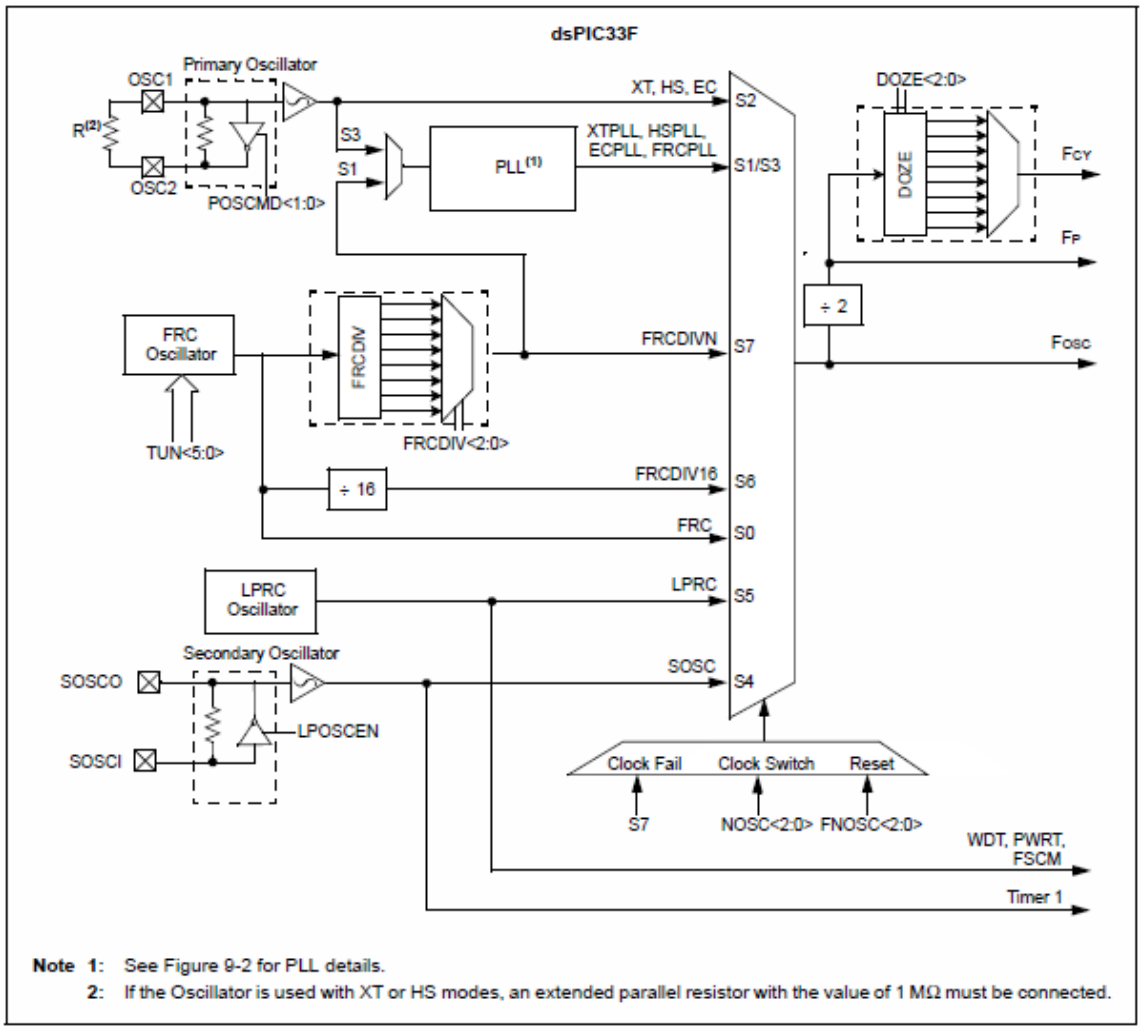


Figura 8.8.2

Aunque este PIC también cuenta con, a parte del oscilador primario, con un oscilador secundario capaz de trabajar a 32,768 kHz, específico para baja potencia o para dedicarlo al Watchdog Timer (WDT).

8.9. PROGRAMACIÓN DEL DSPIC

Para introducir el programa en el microcontrolador son necesarias unas líneas de conexión dedicadas a la programación, y también posible depuración (debbuger), desde el ordenador hasta el PIC. Estas líneas son las denominadas como RB6 y RB7.
Dichas líneas parten de los pines RB6 y RB7 (pines 26 y 27 respectivamente) de este dsPIC, en el que forman parte del puerto B (puerto bidireccional de entrada/salida).

En este diseño, estas líneas están bifurcadas para permitir la programación del microcontrolador tanto mediante el Pickit2 como mediante el ICD2 (programador oficial de Microchip).

8.10. TRANSCEIVER

Un transceiver es un elemento o dispositivo dedicado a la adaptación de señales entre dos sistemas para permitir la comunicación entre ellos. Estos dispositivos sirven como interfaz entre el bus CAN y el controlador con protocolo CAN del microcontrolador. Básicamente, se encarga de convertir las señales digitales a señales ajustadas para la transmisión en el cableado del bus. También proporciona una serie de protecciones (altas tensiones, EMIs, ESDs, transitorios eléctricos, etc) en la comunicación.
Todo dispositivo que utilice un bus CAN requiere de un transceiver para su uso.

En este diseño se ha utilizado el transceiver MCP2551, el cual permite los siguientes modos de operación:

- High-Speed

El modo *High-Speed* (alta velocidad) se selecciona conectando el pin Rs a Vss. En este modo, los controladores de la salida del transmisor tienen una rápida respuesta en los tiempos de subida y bajada para apoyar las tasas de alta velocidad del bus.

- Slope-Control

El modo *Slope-Control* (control por pendiente) reduce adicionalmente las EMI limitando los tiempos de subida y bajada en las líneas CANH y CANL. La pendiente, ó slew rate (SR), is controlada mediante la conexión de una resistencia externa entre Rs y VOL. La pendiente es proporcional a la corriente de salida en el pin de Rs.

- Standby

El dispositivo puede ser puesto en standby o modo "*sleep*" aplicando a nivel alto en Rs. En este modo, el transmisor se desactiva y el receptor opera con corrientes más pequeñas. El pin de recepción en el lado del controlador (RXD) esta todavía funcional pero operará con señales mas pequeñas. El microcontrolador asociado puede monitorizar la actividad del pin RXD del bus CAN y poner al transceiver en modo normal mediante el pin Rs (aunque a mayores señales en el bus, el primer mensaje se puede llegar a perder).

En la siguiente figura se muestra el diagrama de bloques de este transceiver MCP2551:

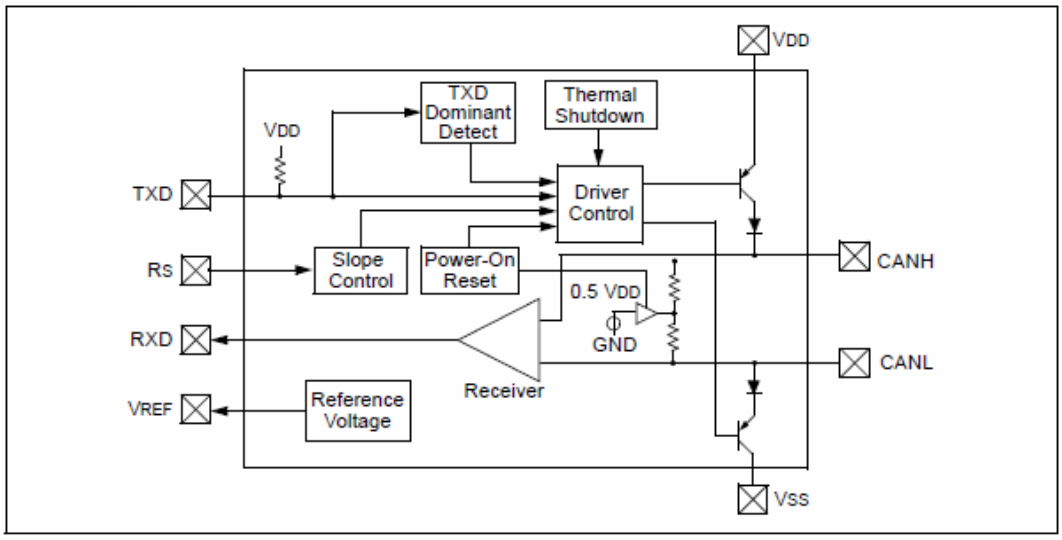


Figura 8.10.1

Para el diseño se ha escogido un encapsulado SOIC.

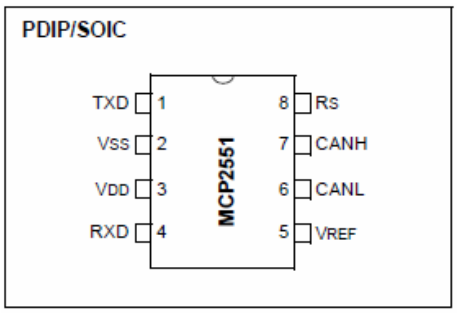
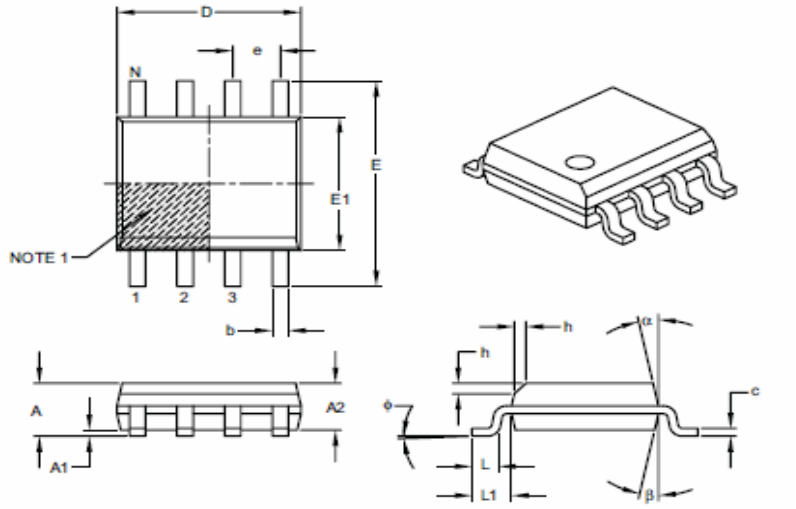


Figura 8.10.2

En la siguiente figura se muestran las dimensiones del componente utilizado:



Units		MILLMETERS		
Dimension Limits		MIN	NOM	MAX
Number of Pins	N		8	
Pitch	e		1.27 BSC	
Overall Height	A	–	–	1.75
Molded Package Thickness	A2	1.25	–	–
Standoff §	A1	0.10	–	0.25
Overall Width	E		6.00 BSC	
Molded Package Width	E1		3.90 BSC	
Overall Length	D		4.90 BSC	
Chamfer (optional)	h	0.25	–	0.50
Foot Length	L	0.40	–	1.27
Footprint	L1		1.04 REF	
Foot Angle	φ	0°	–	8°
Lead Thickness	c	0.17	–	0.25
Lead Width	b	0.31	–	0.51
Mold Draft Angle Top	α	5°	–	15°
Mold Draft Angle Bottom	β	5°	–	15°

Figura 8.10.3

8.11. MEMORIA FLASH

La memoria flash es una tecnología de almacenamiento derivada de la memoria EEPROM que permite la lectura/escritura de múltiples posiciones de memoria en la misma operación. Por lo que la tecnología flash, siempre mediante impulsos eléctricos, permite velocidades de funcionamiento muy superiores frente a la tecnología EEPROM primigenia, que sólo permitía actuar sobre una única celda de memoria en cada operación de programación.
Flash, como tipo de EEPROM que es, contiene una matriz de celdas con un transistor evolucionado con dos puertas en cada intersección. Tradicionalmente sólo almacenan un bit de información. Las nuevas memorias flash, llamadas también dispositivos de celdas multi-nivel, pueden almacenar más de un bit por celda variando el número de electrones que almacenan.
Estas memorias están basadas en el transistor FAMOS (Floating Gate Avalanche-Injection Metal Oxide Semiconductor) que es, esencialmente, un transistor NMOS con un conductor (basado en un óxido metálico) adicional localizado o entre la puerta de control (CG – Control Gate) y los terminales fuente/drenador contenidos en otra puerta (FG – Floating Gate) o alrededor de la FG conteniendo los electrones que almacenan la información.

Es decir, que las tarjetas de memoria flash están hechas de celdas microscópicas que acumulan electrones con diferentes voltajes a medida que la electricidad pasa a través de ellas, creando así un mapa de diferentes cargas eléctricas. De este modo la tarjeta logra guardar la información que el usuario requiere. Mientras más compacta esté distribuida su estructura, mayor información almacena, sin embargo también aumentan los costos en la fabricación de estos dispositivos.

Este tipo de memoria está fabricado con puertas lógicas NOR y NAND para almacenar los 0s ó 1s correspondientes.

Comparativa de memorias flash basadas en NOR y flash basadas en NAND:

- La densidad de almacenamiento de los chips es actualmente bastante mayor en las memorias NAND.
- El coste de NOR es mucho mayor.
- El acceso NOR es aleatorio para lectura y orientado a bloques para su modificación. Sin embargo, NAND ofrece tan solo acceso directo para los bloques y lectura secuencial dentro de los mismos.
- En la escritura de NOR podemos llegar a modificar un solo bit. Esto destaca con la limitada reprogramación de las NAND que deben modificar bloques o palabras completas.
- La velocidad de lectura es muy superior en NOR (50-100 ns) frente a NAND (10 µs de la búsqueda de la página + 50 ns por byte).
- La velocidad de escritura para NOR es de 5 µs por byte frente a 200 µs por página en NAND.
- La velocidad de borrado para NOR es de 1 ms por bloque de 64 KB frente a los 2 ms por bloque de 16 KB en NAND.
- La fiabilidad de los dispositivos basados en NOR es realmente muy alta, es relativamente inmune a la corrupción de datos y tampoco tiene bloques erróneos frente a la escasa fiabilidad de los sistemas NAND que requieren corrección de datos y existe la posibilidad de que queden bloques marcados como erróneos e inservibles.

En resumen, los sistemas basados en NAND son más baratos y rápidos pero carecen de una fiabilidad que los haga eficientes, lo que demuestra la necesidad imperiosa de un buen sistema de archivos. Dependientemente de qué sea lo que se busque, merecerá la pena decantarse por uno u otro tipo.

Si estos elementos no sustituyen a los discos duros, a parte de por las características mencionadas antes, es debido a que el coste por MB de la memoria flash es muy superior a los discos duros, y además los discos duros tienen una capacidad muy superior a la de las memorias flash.

En este diseño se ha añadido la memoria flash M25P80, que añade las siguientes características:

- Memoria flash de 8 Mbit
- Borrado de 512kbit en 2 segundos
- Tension de alimentación de 2,7V a 3,6V
- Compatible con bus SPI a 25 MHz
- Mas de 100.000 borrados
- Retención de datos superior a 20 años

En la siguiente figura se puede ver su diagrama de pines:

C	Serial Clock
D	Serial Data Input
Q	Serial Data Output
$\overline{S}$	Chip Select
$\overline{W}$	Write Protect
$\overline{HOLD}$	Hold
V_{CC}	Supply Voltage
V_{SS}	Ground

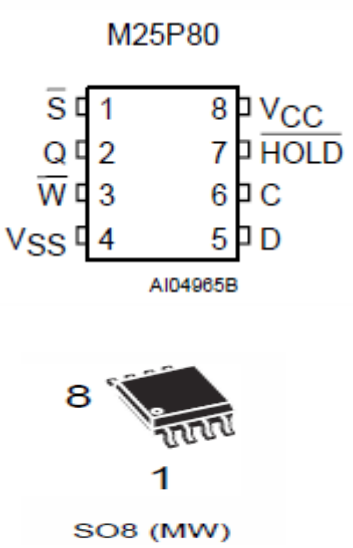


Figura 8.11.1

La memoria interna de esta memoria flash está organizada según:

- 1.048.576 bytes (de 8 bits cada uno)
- 16 sectores (de 512 Kbits, 65536 bytes cada uno)
- 4096 páginas (256 bytes cada una)

Sector	Address Range	
15	F0000h	FFFFFh
14	E0000h	EFFFFh
13	D0000h	DFFFFh
12	C0000h	CFFFFh
11	B0000h	BFFFFh
10	A0000h	AFFFFh
9	90000h	9FFFFh
8	80000h	8FFFFh
7	70000h	7FFFFh
6	60000h	6FFFFh
5	50000h	5FFFFh
4	40000h	4FFFFh
3	30000h	3FFFFh
2	20000h	2FFFFh
1	10000h	1FFFFh
0	00000h	0FFFFh

Donde el conjunto de instrucciones útiles para interactuar con el componente son:

Instruction	Description	One-byte Instruction Code	Address Bytes	Dummy Bytes	Data Bytes
WREN	Write Enable	0000 0110	0	0	0
WRDI	Write Disable	0000 0100	0	0	0
RDSR	Read Status Register	0000 0101	0	0	1 to ∞
WRSR	Write Status Register	0000 0001	0	0	1
READ	Read Data Bytes	0000 0011	3	0	1 to ∞
FAST_READ	Read Data Bytes at Higher Speed	0000 1011	3	1	1 to ∞
PP	Page Program	0000 0010	3	0	1 to 256
SE	Sector Erase	1101 1000	3	0	0
BE	Bulk Erase	1100 0111	0	0	0
DP	Deep Power-down	1011 1001	0	0	0
RES	Release from Deep Power-down, and Read Electronic Signature	1010 1011	0	3	1 to ∞
	Release from Deep Power-down		0	0	0

En la siguiente figura está dibujado el diagrama de bloques del M25P80:

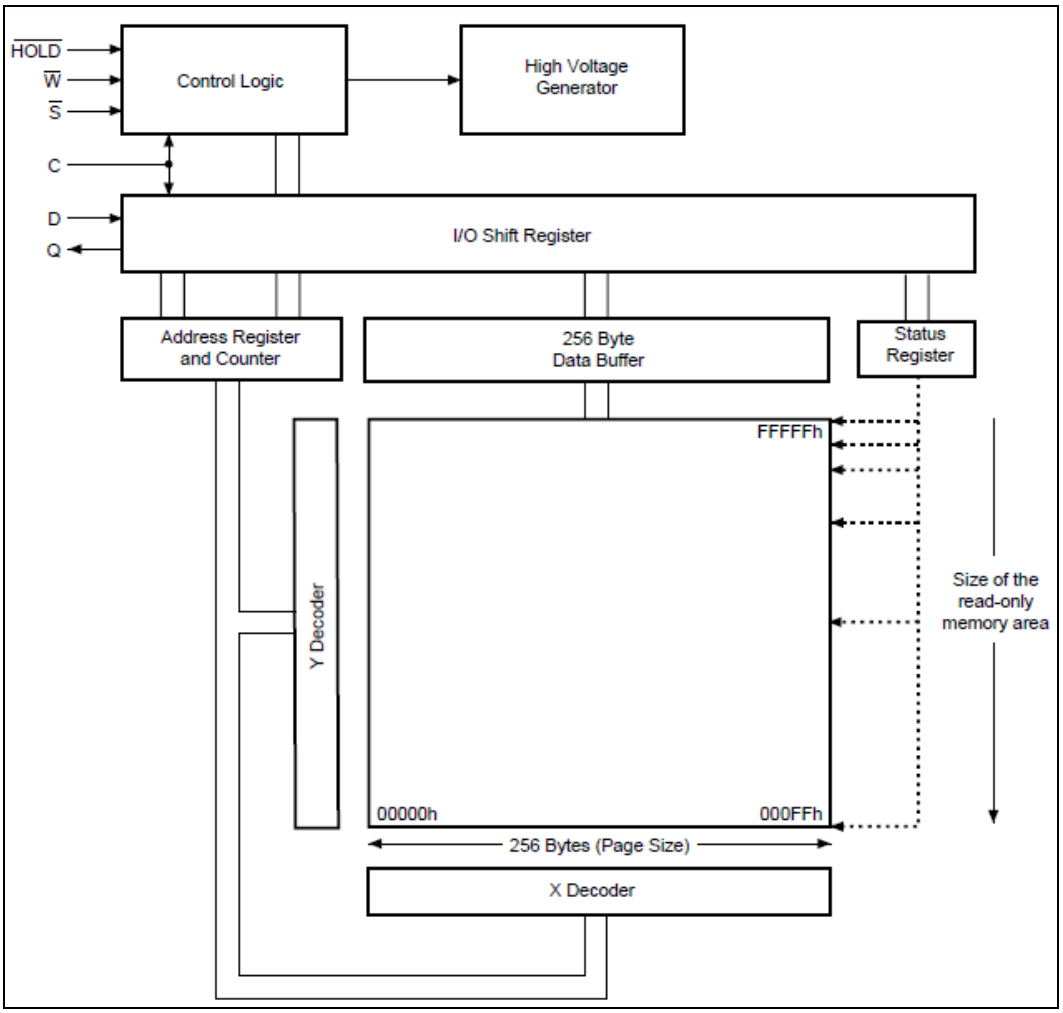


Figura 8.11.2

En este diagrama está recogida toda la información que antes se mencionaba, lo cual da una idea más precisa de cómo es el funcionamiento interno de este elemento así como de la distribución de los datos y su gestión interna.

Además, para proteger la información contenida, presenta en su *datasheet* un sistema de protección, tanto software como hardware:

$\overline{W}$ Signal	SRWD Bit	Mode	Write Protection of the Status Register	Memory Content	
				Protected Area	Unprotected Area
1	0	Software Protected (SPM)	Status Register is Writable (if the WREN instruction has set the WEL bit) The values in the SRWD, BP2, BP1 and BP0 bits can be changed	Protected against Page Program, Sector Erase and Bulk Erase	Ready to accept Page Program and Sector Erase instructions
0	0				
1	1				
0	1	Hardware Protected (HPM)	Status Register is Hardware write protected The values in the SRWD, BP2, BP1 and BP0 bits cannot be changed	Protected against Page Program, Sector Erase and Bulk Erase	Ready to accept Page Program and Sector Erase instructions

Figura 8.11.3

8.12. TARJETA SD

Las tarjetas SD son un tipo de memorias flash, es decir, no volátiles, por lo que no necesitan estar alimentadas para conservar sus datos.
Todas las tarjetas de memoria SD y SDIO necesitan soportar el modo SPI/MMC que soporta la interfaz serie SPI de cuatro cables ligeramente lenta (reloj, entrada serial, salida serial y selección de chip), compatible con los puertos SPI de muchos microcontroladores.

Ventajas:

- Gran capacidad de almacenamiento
- Interfaz física/eléctrica sencilla
- Gran disponibilidad en el mercado
- Reducido tamaño en formato microSD
- Son portables
- Costos reducidos.
- Rango de alimentación de 2,7 V a 3,6 V

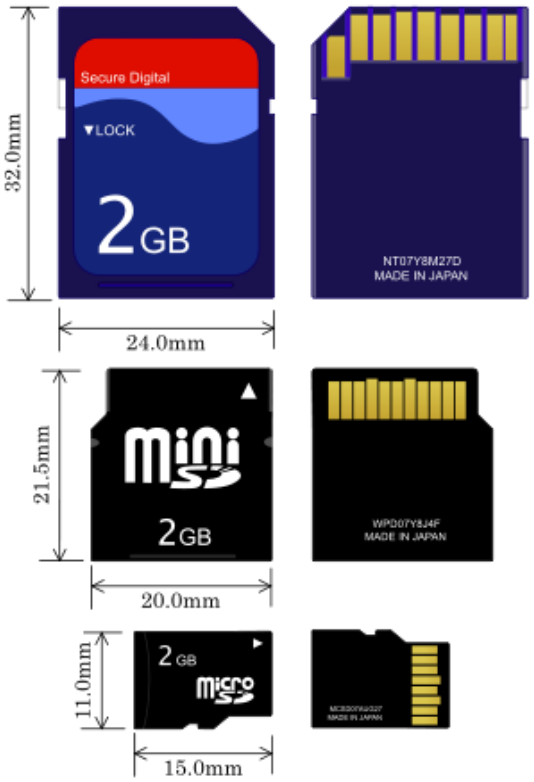


Figura 8.12.1

La tarjeta SD no es el único estándar de tarjetas de memoria flash ratificado por la Secure Digital Card Association. Existen otros formatos de dicha asociación, como son el MiniSD y el MicroSD (conocido como Transflash antes de la ratificación por la Secure Digital Card Association).
Estas tarjetas más pequeñas se pueden utilizar en ranuras del mismo tamaño que MMC/SD/SDIO con un adaptador (que deba conectar las conexiones eléctricas así como llevar a cabo el contacto físico). Sin embargo, hay que decir que ya es difícil crear dispositivos de E/S con el factor de forma del SD y esto será aún menos posible con tamaños más pequeños. Como las ranuras SD todavía tienen soporte para las tarjetas MMC, las variantes de MMC más pequeñas, que han evolucionado, también son compatibles con los dispositivos lectores para SD.

El límite de la capacidad de todos los formatos SD/MMC parece ser de 128 GB en modo LBA (direccionamiento de sector de 28 bits).

Actualmente todas las tarjetas fabricadas por SanDisk, Ritek/Ridata y Kingmax Digital utilizan el modo SPI. Además, las tarjetas MMC pueden ser eléctricamente idénticas a las tarjetas SD pero en una carcasa más fina y con un fusible para deshabilitar las funciones de SD (así que no es necesario pagar royalties SD).

El protocolo subyacente SPI ha existido durante años como una característica estándar en muchos microcontroladores. Mientras SPI utilizaba tres líneas compartidas más una de selección de chip separada en cada tarjeta, el nuevo protocolo (un-bit MMC/SD), que utiliza la señalización del colector abierto para permitir múltiples tarjetas en el mismo bus (pero que causa problemas a frecuencias de reloj altas), permite que hasta 30 tarjetas sean conectadas con los mismos tres cables (sin la selección de chip) a expensas de una inicialización mucho más complicada de la tarjeta y del requisito de que cada tarjeta tiene un número de serie único para el conector y debe solicitar autorización para realizar la operación. Esta característica se utiliza raramente, y su uso se desaconseja en los nuevos estándares (que recomiendan un canal totalmente separado a cada tarjeta) debido a cuestiones de velocidad y consumo de energía. El protocolo cuasi-propietario de un-bit MMC/SD fue ampliado para utilizar transferencias con un ancho de cuatro bits (SD y MMC) y ocho bits (sólo MMC) para lograr más velocidad, mientras que la mayor parte del resto de la industria de la informática se está trasladando a canales más estrechos de una velocidad más alta.

El estándar SPI se habría podido registrar simplemente con unas frecuencias de transferencia de datos más altas (por ejemplo, 133 MHz) para tener un rendimiento más alto que el ofrecido por el SD de cuatro bits (de todos modos, las CPU embebidas que ya no tenían tasas de reloj más altas no habrían sido lo suficientemente rápidas como para manejar tasas de datos más altas). La asociación de la tarjeta SD dio soporte para parte de las órdenes del antiguo protocolo MMC de un bit y añadió soporte para comandos adicionales relacionados con la protección de copia.

En la siguiente figura se muestran los diagramas de pines de las tarjetas MMC y SD:

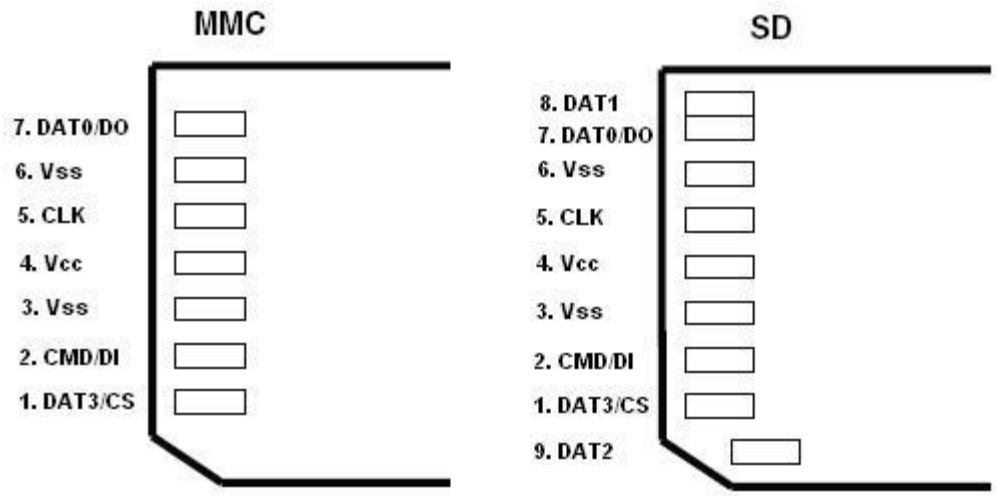


Figura 8.12.2

Orientando la información hacía esta tarjeta SD en modo SPI:

PIN_No	Nombre	Descripción
1	CS	Activación de tarjeta
2	DATA IN	Comandos de datos desde el host
3	VSS	GND
4	VDD	Alimentación
5	CLK	Reloj
6	VSS	GND
7	DATAOUT	Datos hacia el Host
8	RSV	Reservado
9	RSV	Reservado

La memoria contiene varios comandos para SPI, incluyendo:

- Comando 0: Reset de la memoria (CMD0)
- Comando 1: Inicialización de la memoria (CMD1)
- Comando 16: Configuración del bloque de 512 bytes (CMD16)
- Comando 24: Escritura de un bloque de 512 bytes (CMD24)
- Comando 17: Lectura de un bloque de 512 bytes (CMD17)

Una vez enviado el respectivo comando, la memoria responde por medio de un registro llamado R1, indicando si hubo algún error:

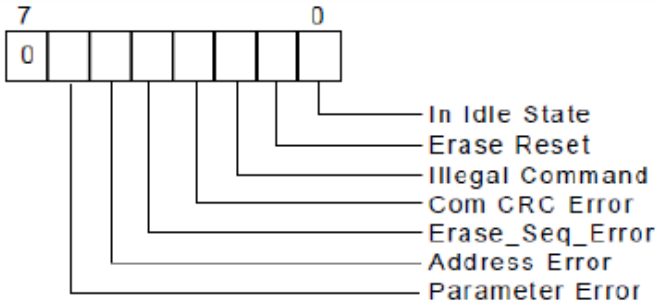


Figura 8.12.3

NOTA: La memoria debe responder 0x00 para todos los comandos antes mencionados, menos para el comando 0 que debe responder con 0x01 debido a su estado inicial.

Iniciar en modo SPI la memoria SD:

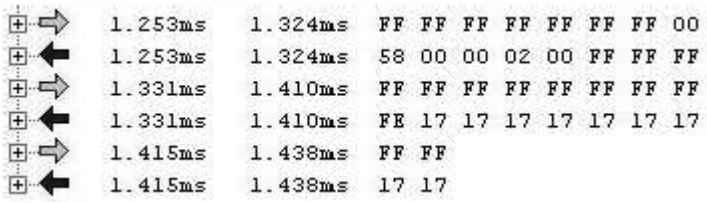
Para que la memoria pueda escribir o leer un sector determinado primero se debe inicializar en modo SPI, es un paso fundamental en el manejo de la memoria, además para que se inicialice de forma correcta se debe enviar el comando CMD0, CMD1 y CMD16 sin activar el pin C2 (0V).
Para inicializar la memoria el PIC debe enviar los seis bytes de comando CMD0, los seis bytes del CMD1 y los seis bytes del CMD16.
Esto se debe a que al iniciar la memoria se establece en modo SD, por lo que para entrar en modo SPI se debe enviar el CMD0 con el pin CS a 0V, tal que si la memoria reconoce la petición de cambio de protocolo responde en R1 con 0x01.

Escritura/Lectura de datos:

Estas operaciones se realizan enviando el comando correspondiente junto a la dirección del primer byte del bloque con el largo indicado anteriormente (CDM16). El largo del bloque puede ser desde 1 hasta 512 bytes, la dirección para realizar la operación debe ser la del byte inicial del sector.

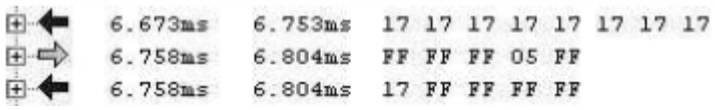
Para realizar escritura de un único bloque se debe enviar el comando CMD24 (0x58, 0x00, 0x00, 0x02, 0x00, 0xFF) indicando la dirección del bloque en el argumento de la función. La memoria al reconocer el comando envía la respuesta R1, donde se indica si se ha producido algún error. Si todo es correcto, el PIC debe enviar un *token* (0xFE) y luego los 512 bytes de datos del bloque más 2 bytes de CRC.
Tras el envío de los datos, se quedará a la espera de una respuesta de la memoria (para ver si se han recibido correctamente). Si no hubo error la memoria responde con 0x05.

Véanse estos pasos en un seguimiento más gráfico:



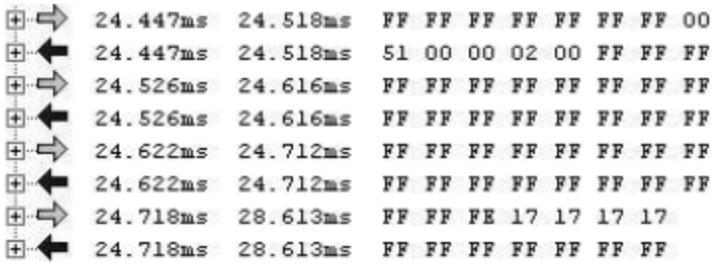
Obsérvese como el microcontrolador envía la cadena de 6 bytes (0x58, 0x00, 0x00, 0x02, 0x00, 0xFF). Es importante notar que el argumento indica que se va a escribir en el segundo bloque de la memoria. También se puede apreciar como la memoria responde con 0x00 a la petición de escritura y después el micro envía el *token* 0xFE y los 512 bytes.

En la siguiente figura se ve que una vez escritos los 512 bytes de datos, el micro envía dos 0xFF y después la memoria responde con 0x05.



Nótese que las respuestas no son inmediatas, donde los bytes FF representan tiempos de espera.

Para realizar una lectura se debe enviar el comando CMD17 (0x51, 0x00, 0x00, 0x02, 0x00, 0xFF) indicándose en el argumento la dirección del bloque. Luego se esperará la respuesta de R1 por parte de la memoria. Si todo es correcto se recibirá el *token* 0xFE y luego los datos, donde la cantidad esta establecida por el largo del bloque (CMD16).



Se puede observar observar como se envía el comando CMD17 y como la memoria responde con 0x00. Después de algunos pulsos de reloj la memoria envía el *token* 0xFE indicando que va a enviar los datos y posteriormente se los envía al microcontrolador.

8.13. ALIMENTACIÓN

La alimentación de una placa de circuito impreso no es una parte trivial del diseño, sino posiblemente la más importante, ya que esta alimentación influye en toda la circuitería empleada en la tarjeta. En este diseño se requiere una alimentación de tensión continua de 3,3V obtenida a través de una batería de 12V.

Debido a que un buen sistema de alimentación requiere estabilidad, rendimiento, ausencia de ruido y unos niveles de tensión ajustados a las especificaciones de los elementos del diseño. Por lo que para conseguir esto, en este diseño se ha optado por hacer un primer ajuste de la tensión a 5V (para multiplicar las posibilidades de nuestra tarjeta) y otro ajuste posterior a 3,3V para el propio funcionamiento de la tarjeta.

El primer ajuste se realiza mediante el LM2576T, capaz de proporcionar una corriente de salida de hasta 3 amperios.

Su diagrama de bloques es el siguiente:

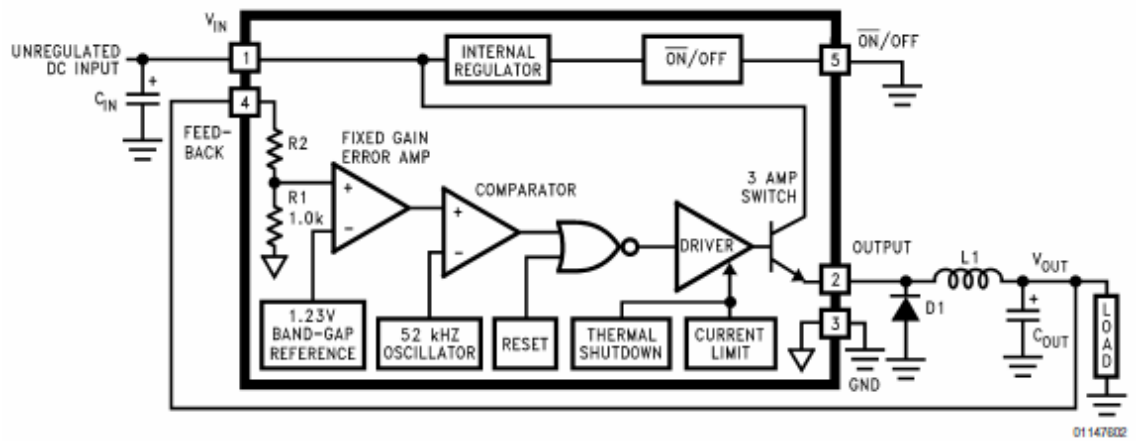


Figura 8.13.1

Su diagrama de pines es (encapsulado de inserción):

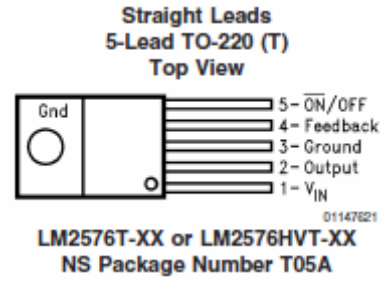
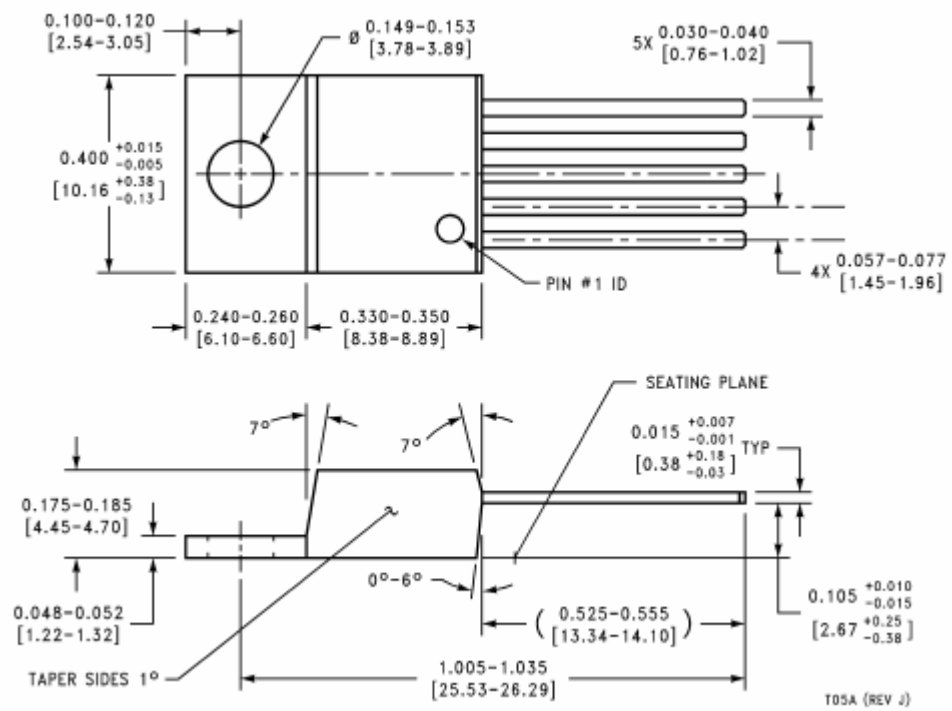


Figura 8.13.2

Dimensiones físicas del LM2576T:



NOTA: Pulgadas (milímetros)

Figura 8.13.3

El segundo ajuste se realiza mediante el LM1117T, capaz de proporcionar una corriente de salida de 800 miliamperios.

El diagrama de bloques es el siguiente:

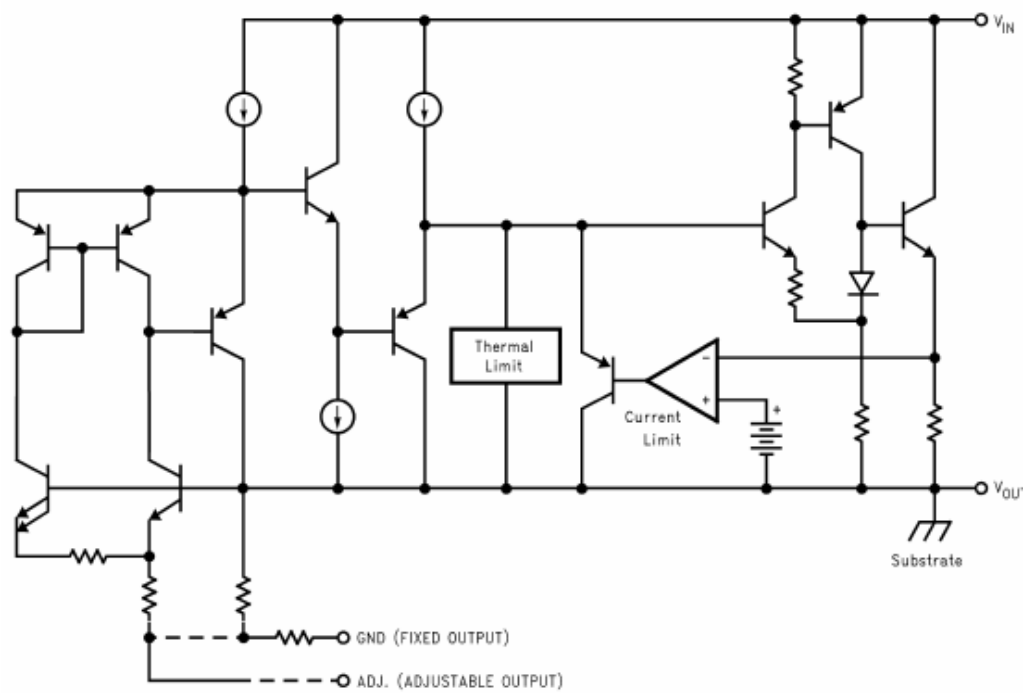


Figura 8.13.4

Su diagrama de pines es (encapsulado SMD):

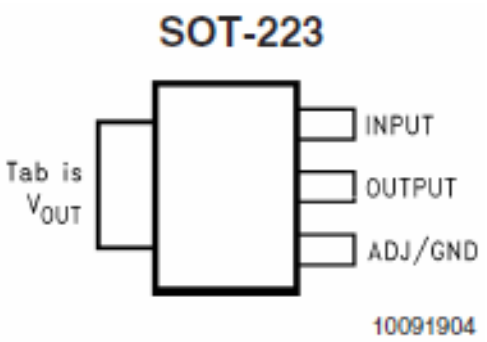


Figura 8.13.5

Dimensiones físicas del LM1117T:

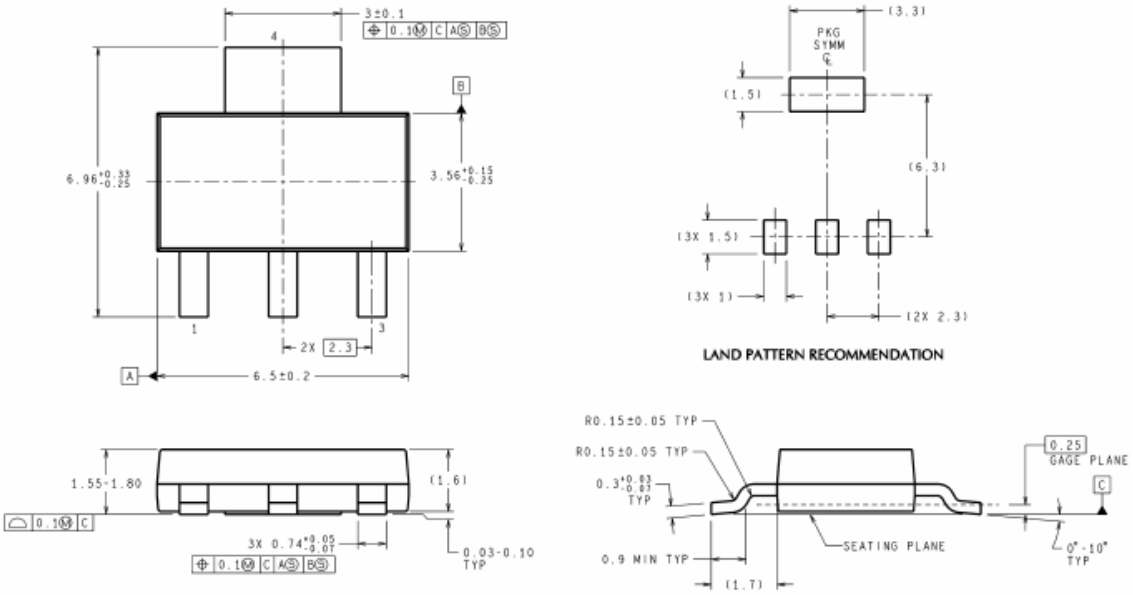


Figura 8.13.6

8.14. SENSORES

Un sensor es un dispositivo capaz de detectar magnitudes físicas o químicas, llamadas variables de instrumentación, y transformarlas en variables eléctricas. Las variables de instrumentación pueden ser por ejemplo: temperatura, intensidad lumínica, distancia, aceleración, inclinación, desplazamiento, presión, fuerza, torsión, humedad, pH, etc.
Una magnitud eléctrica puede ser una resistencia eléctrica (como en una RTD), una capacidad eléctrica (como en un sensor de humedad), una tensión eléctrica (como en un termopar), una corriente eléctrica (como en un fototransistor), etc.
Un sensor se diferencia de un transductor en que el sensor está siempre en contacto con la variable de instrumentación con lo que puede decirse también que es un dispositivo que aprovecha una de sus propiedades con el fin de adaptar la señal que mide para que la pueda interpretar otro dispositivo. Como, por ejemplo, el termómetro de mercurio que aprovecha la propiedad que posee el mercurio de dilatarse o contraerse por la acción de la temperatura.
Es decir, un sensor es un dispositivo que convierte una forma de energía en otra.

Para determinar la calidad de un sensor, se debe tener en cuenta que la resolución de un sensor es el menor cambio en la magnitud de entrada que se aprecia en la magnitud de salida. Sin embargo, la precisión es el máximo error esperado en la medida.
Esto significa que la resolución puede ser de menor valor que la precisión. Por ejemplo, si al medir una distancia la resolución es de 0,01 mm, pero la precisión es de 1 mm, entonces pueden apreciarse variaciones en la distancia medida de 0,01 mm, pero no puede asegurarse que haya un error de medición menor a 1 mm. En la mayoría de los casos este exceso de resolución conlleva a un exceso innecesario en el coste del sistema.
No obstante, si el error en la medida sigue una distribución normal o similar, lo cual es frecuente en errores accidentales, es decir, no sistemáticos, la repetitividad podría ser de un valor inferior a la precisión.
Sin embargo, la precisión no puede ser de un valor inferior a la resolución, pues no puede asegurarse que el error en la medida sea menor a la mínima variación en la magnitud de entrada que puede observarse en la magnitud de salida.

Características de un sensor:

- Rango de medida: Intervalo que domina el sensor para realizar una medida.
- Precisión: Es el error de medida máximo esperado.
- Offset (ó desviación de cero): Valor de la variable de salida cuando la variable de entrada es nula. Si el rango de medida no llega a valores nulos de la variable de entrada, habitualmente se establece otro punto de referencia para definir el Offset.
- Linealidad (ó correlación lineal): Es la capacidad del elemento para proporcionar una relación lineal (de línea recta) con una magnitud determinada distinta de una magnitud de influencia.
- Sensibilidad: Relación del sensor entre la variación de la magnitud de salida y la variación de la magnitud a la entrada.
- Resolución: Mínima variación de la magnitud de entrada que puede apreciarse a la salida.
- Rapidez de respuesta: Puede ser un tiempo fijo o depender de la variable a medir. Es la capacidad del sistema para responder a la salida ante las variaciones en la magnitud de entrada.
- Derivas: Son unas magnitudes, aparte de la medida a la entrada, que influyen en la variable de salida (condiciones ambientales como humedad, temperatura, también afectan: envejecimiento, oxidación, desgaste, etc).
- Repetitividad: Error esperado al repetir varias veces la misma medida.

En este diseño se emplea un sensor de temperatura que mide mediante rayos infrarrojos, el cual se denomina: termopila MLX90614.

En un principio, se planteó el uso de este tipo de sensores con una salida analógica en su salida, pero fueron finalmente descartados por la comodidad de no tener que añadir una parte analógica, aprovechándose la evolución digital de dichos sensores. La decisión de sustituir el elemento analógico (MLX90247) por el digital de salida PWM (MLX90614) se debió a:

- Una salida PWM no requiere de una etapa de acondicionamiento de la señal, lo cual reduce la complejidad del diseño hardware.
- Un sensor PWM permite conectar varios elementos al mismo bus I^2C, permitiendo aumentar la densidad de integración en la PCB.

Sin embargo, aportan la desventaja económica, ya que son considerablemente más caros. Pero ya que las ventajas que aportan son convincentes, finalmente fueron los sensores PWM los añadidos al diseño.

En este proyecto se ha añadido el sensor MLX90614, específico para medidas de temperatura sin contacto con el objeto a medir. Dicho elemento tiene un chip detector sensible a los infrarrojos y el acondicionamiento de la señal integrados en el mismo encapsulado TO-39.

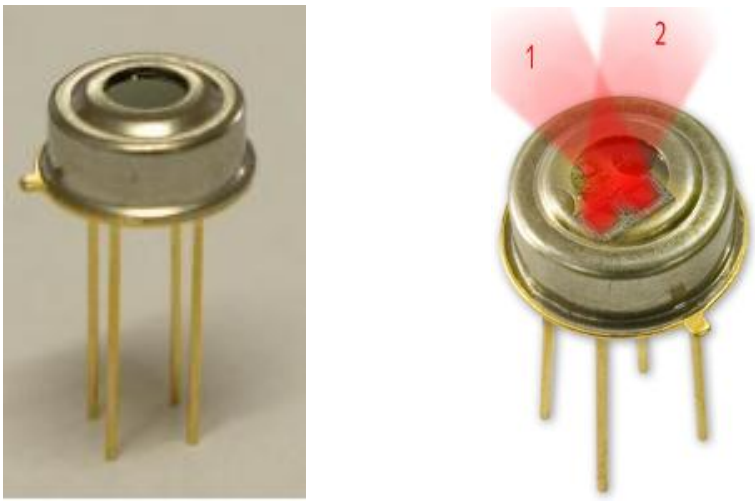


Figura 8.14.1

Gracias a su amplificador de bajo ruido, su ADC de 17-bits y su unidad DSP, consigue una alta precisión y resolución en las medidas.
Este componente viene calibrado de fábrica con un PWM digital y una salida SMBus (System Management Bus). Predefinido para que el PWM de 10 bits esté continuamente transmitiendo las medidas de temperatura dentro del rango -20º a 120ºC, con una resolución de salida de 0.14ºC.
La configuración del POR por defecto de fábrica esta dedicada para el SMBus.

Los dos chips integrados que contiene, son desarrollados y fabricados por Melexis:

- La termopila detectora de infrarrojos MLX81101.
- Y el acondicionador de señal ASSP MLX90302, especialmente diseñado para procesar la salida del sensor IR.

Diagrama de bloques:

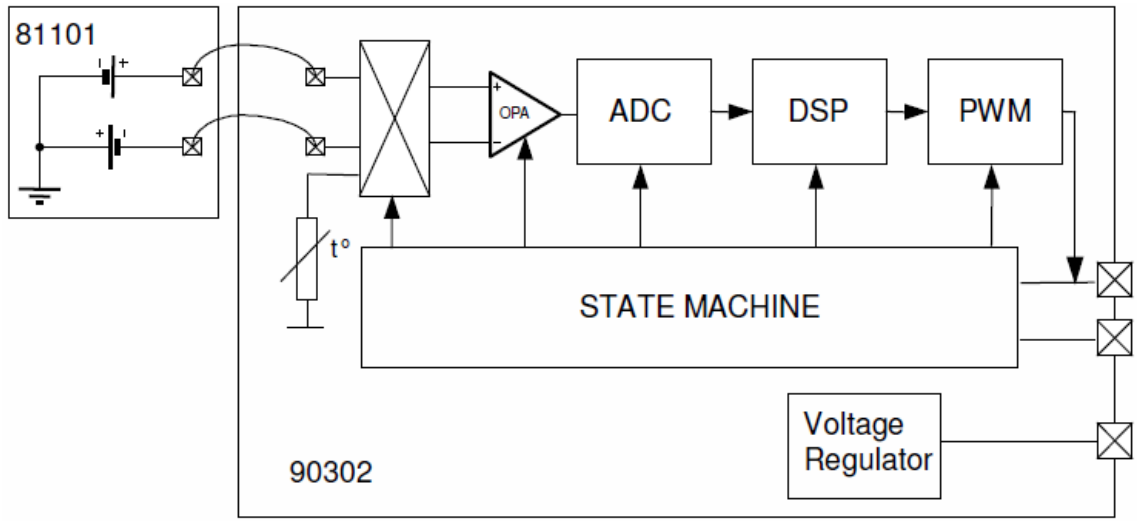


Figura 8.14.2

La operación del MLX90614 es controlada por un autómata (state machine), el cual controla las medidas y cálculos del objeto a medir y de la temperatura ambiente, haciendo un post-procesamiento de las temperaturas de salida a través del PWM ó de la interfaz SMBus.

La salida del sensor IR es amplificada por el amplificador de bajo ruido con corrección de Offset y ganancia programable que integra, convertida dicha salida por un modulador Sigma-Delta a formato digital (una cadena de bits única) y llevada a un DSP para un procesamiento adicional. La señal es tratada por filtros (paso bajo) FIR e IIR para una mayor reducción del ancho de banda de la señal de entrada y así alcanzar el rendimiento deseado en cuanto a ruido y a las tasas de actualización de la medida. Como último paso del ciclo de medición, las medidas Ta y To son reescaladas a la resolución de salida deseada, y también los datos recalculados son cargados en el registro "estado de maquina" del PWM, el cual crea una constante de frecuencia con un ciclo de trabajo representativo de los datos medidos.

Diagrama de pines del sensor:

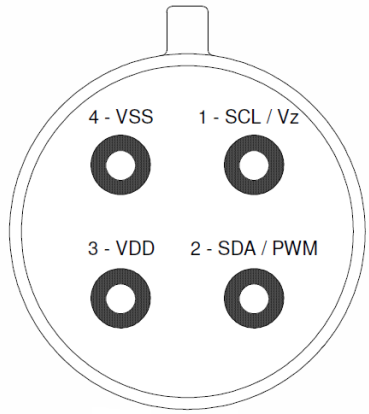


Figura 8.14.3

- SCL/Vz: Entrada serie de reloj para protocolos de comunicaciones de dos cables. Un diodo zener de 5,7V esta disponible en este pin para la conexión externa de un transistor bipolar al MLX90614A para suministrarle al dispositivo desde 8V hasta 16V desde una fuente externa.
- SDA/PWM: Entrada/Salida digital. En modo normal, la temperatura medida esta disponible en este pin mediante PWM (Pulse Width Modulated). En modo compatible con SMBus es configurado automáticamente como drenador abierto NMOS.
- VDD: Tensión de alimentación externa. Puede ser: 3,3V, 5V, ó de 8V a 16V.
- VSS: Masa. El metal está también conectado a este pin.

Registros de la memoria EEPROM interna útiles a utilizar:

EEPROM (32X16)		
Name	Address	Write access
To_{max}	0x000	Yes
To_{min}	0x001	Yes
PWMCTRL	0x002	Yes
Ta range	0x003	Yes
Emissivity correction coefficient	0x004	Yes
Config Register1	0x005	Yes
Melexis reserved	0x006	No
...	...	...
Melexis reserved	0x00D	No
SMBus address	0x00E	Yes
Melexis reserved	0x00F	Yes
Melexis reserved	0x010	No
...	...	...
Melexis reserved	0x018	No
Melexis reserved	0x019	Yes
Melexis reserved	0x01A	No
Melexis reserved	0x01B	No
ID number	0x01C	No
ID number	0x01D	No
ID number	0x01E	No
ID number	0x01F	No

Configuración del periodo del PWM:

En modo extendido el periodo es dos veces el periodo del modo simple, tal que en modo simple el periodo es: T = 1.024 * P[ms] (P es el número escrito en bits de 15…9 PWMCTRL). Donde el periodo máximo es 131.072 ms para el modo simple y 262.144 para el modo extendido. Estos son los valores típicos, que dependen del valor absoluto del oscilador RC del on-chip. El ciclo de trabajo debe ser calculado para trabajar sólo con el nivel alto para evitar errores de las desviaciones del valor absoluto del periodo.

Las direcciones del PWMCTRL consisten en bits de control para configurar el pin PWM/SDA como se muestra en la siguiente figura:

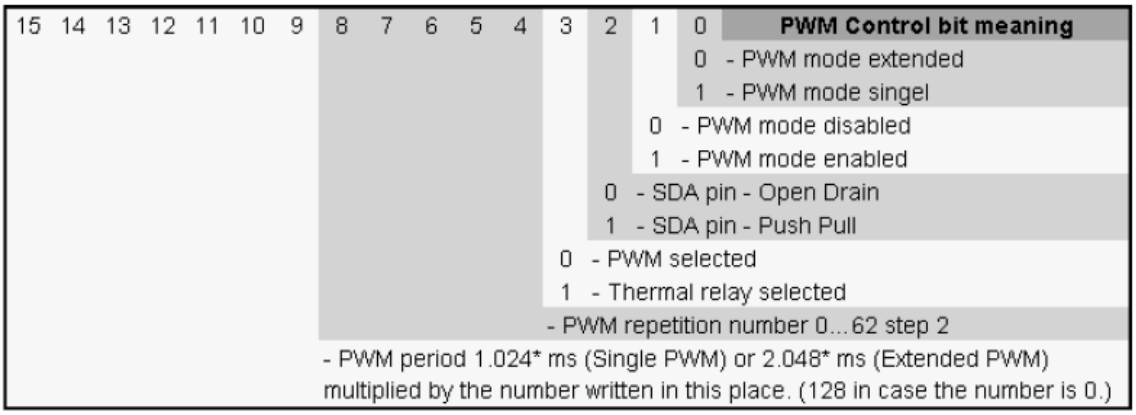


Figura 8.14.4

Registros de la memoria RAM interna útiles a utilizar:

RAM (32x17)		
Name	**Address**	**Read access**
Melexis reserved	0x000	Yes
...	...	...
Melexis reserved	0x003	Yes
Raw data IR channel 1	0x004	
Raw data IR channel 2	0x005	
T_A	0x006	Yes
T_{OBJ1}	0x007	Yes
T_{OBJ2}	0x008	Yes
Melexis reserved	0x009	Yes
...	...	...
Melexis reserved	0x01F	Yes

No es posible escribir en la memoria RAM, de hecho sólo es posible leer de unos pocos y determinados registros, así como se muestra en la tabla anterior.
Apréciese que básicamente se trata de las temperaturas procesadas que ofrece el sensor.

Este sensor permite dos tipos de conexionado con el MCU:

1. Modo de operación de salida PWM.

Es un modo muy sencillo como se observa en la figura 8.14.5. El modo PWM opera libremente tras haber sido configurado el POR de la EEPROM. Se debe forzar al pin SCL a un nivel alto para usar el modo PWM (puede ser unido al pin V_{DD}).

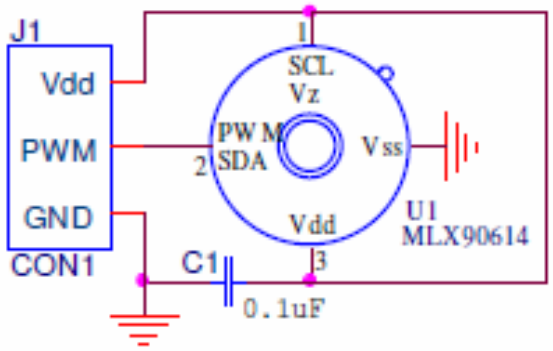


Figura 8.14.5

Una resistencia de pull-up se puede usar para preservar la opción del modo de operación SMBus mientras tenemos el modo PWM por defecto como se muestra a continuación en la figura 8.14.6 que clarifica la conexión comentada.

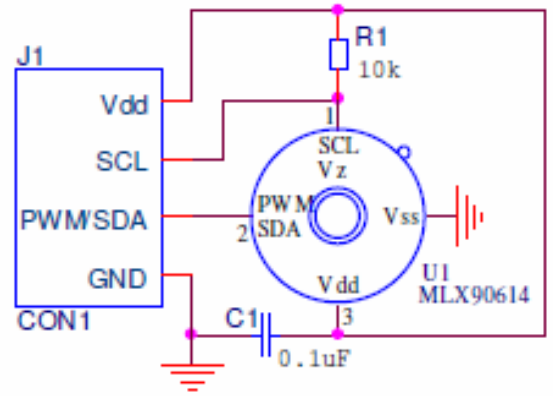


Figura 8.14.6

De nuevo, el modo PWM necesita que escriba en el POR de la EEPROM. Después para la operación en PWM la línea SCL puede estar en alta impedancia (forzada a nivel alto), ó incluso estar sin conectar. La resistencia de pull-up R1 asegurará que haya un nivel

alto en el pin SCL y el PWM POR por defecto será activado. De todos modos el SMBus está todavía disponible (por ejemplo, para una reconfiguración adicional del MLX90614, o para la gestión del modo sleep).
El modo PWM se puede configurar en drenador abierto NMOS o como salida push-pull. En el caso de drenador abierto se requiere de una resistencia externa de pull-up.

2. Modo de operación con configuración en SMBus.

El MLX90614 soporta direcciones de 7 bits en la EEPROM, lo cual permite hasta 127 dispositivos a través de dos cables comunes (lo que permite medir con estos sensores hasta 254 objetos y 127 temperaturas ambiente). Para su funcionamiento necesita de, o bien dos resistencias de pull-up, o bien de dos fuentes de corriente de pull-up (preferibles para altas capacidades parasitas dentro del bus).

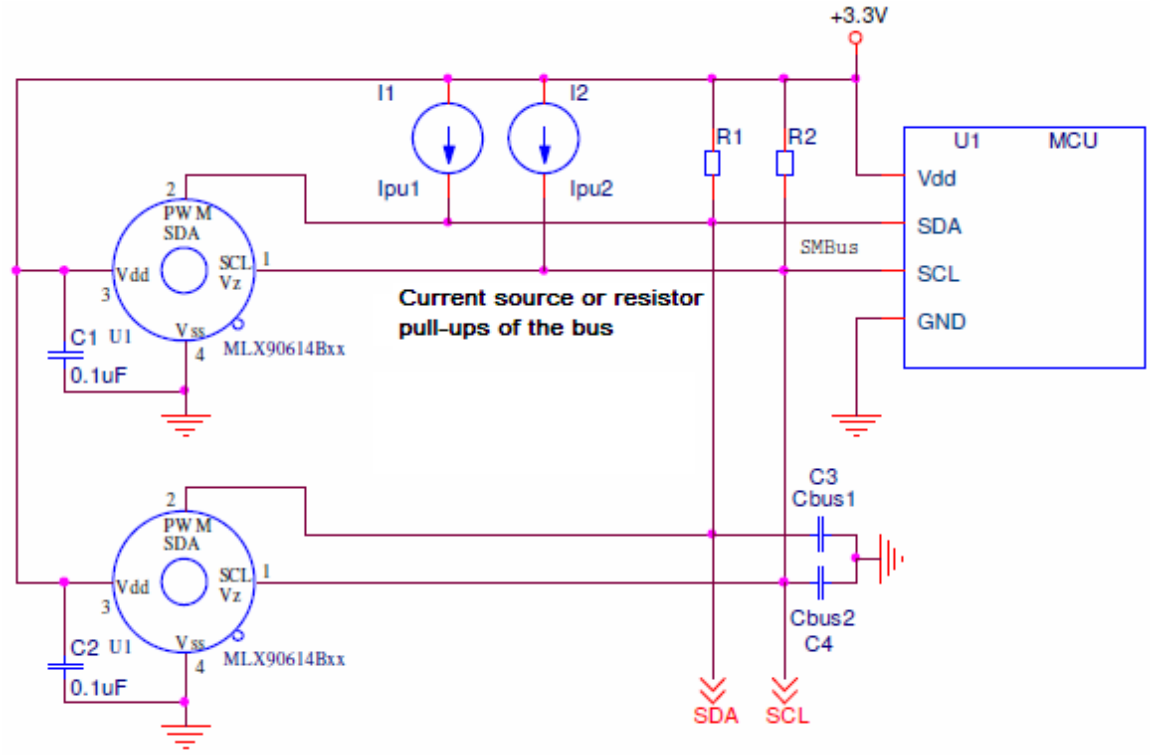


Figura 8.14.7

NOTA: SMBus (Bus de Administración del Sistema) es un subconjunto del protocolo I^2C.

Dimensiones del componente:

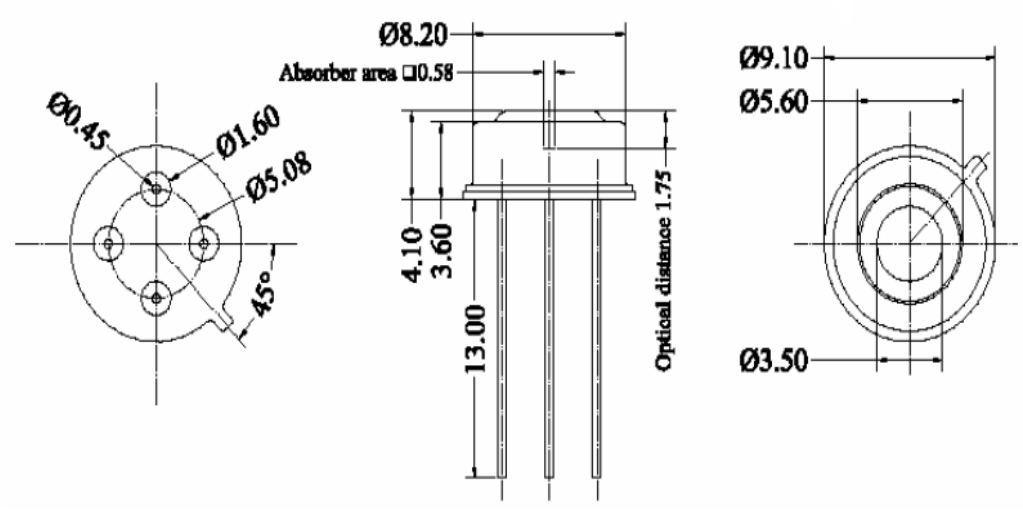


Figura 8.14.8

9. DESARROLLO HARDWARE

En esta parte se detallará el conexionado entre los distintos componentes, así como su desarrollo mediante el programa (o entorno de desarrollo) empleado para tal fin.

9.1. EAGLE

En concreto este proyecto se ha diseñado con el entorno de código abierto EAGLE.

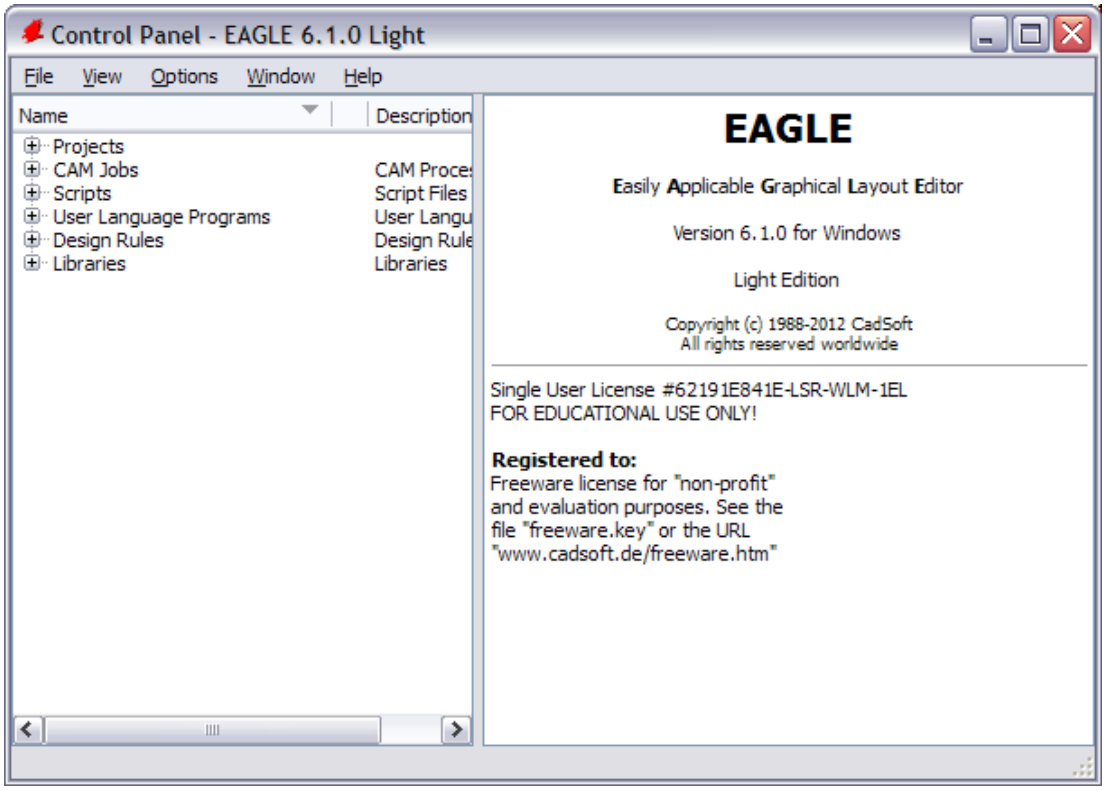


Figura 9.1.1

Se ha escogido Eagle debido a las ventajas que ofrece a la hora de diseñar, ya que muestra para un mismo componente distintos encapsulados que se pueden visualizar de forma gráfica en el momento de establecer el componente en el esquemático del circuito. También porque es un entorno muy extendido, donde Cadsoft diseña las librerías (con sus respectivas huellas) para poder usarlas cómodamente en Eagle adecuándolas a las especificaciones de cada fabricante. Por tanto, se va creando de forma paralela al esquemático (archivo de extensión ".sch") el diseño final orientado a

la PCB (archivo de extensión ".brd"). A su vez, esto permite que se pueda conmutar entre distintos archivos (o ventanas dentro del programa) para ver como se van desarrollando los avances en ambos esquemas y efectuar los cambios sin esperar a finalizar el .sch y con una visión general del diseño mucho más clara y previsora.

Con respecto a lo comentado anteriormente:

- Distintas huellas para un mismo componente

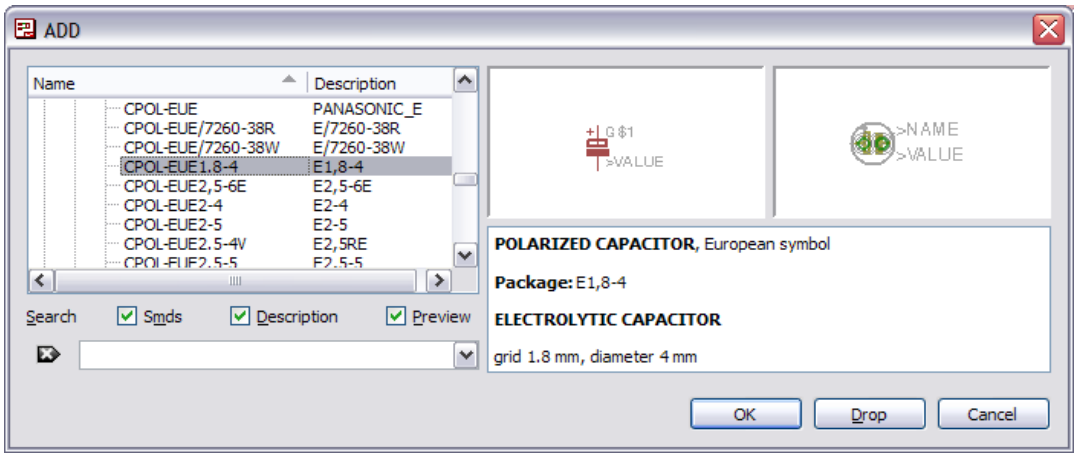


Figura 9.1.2

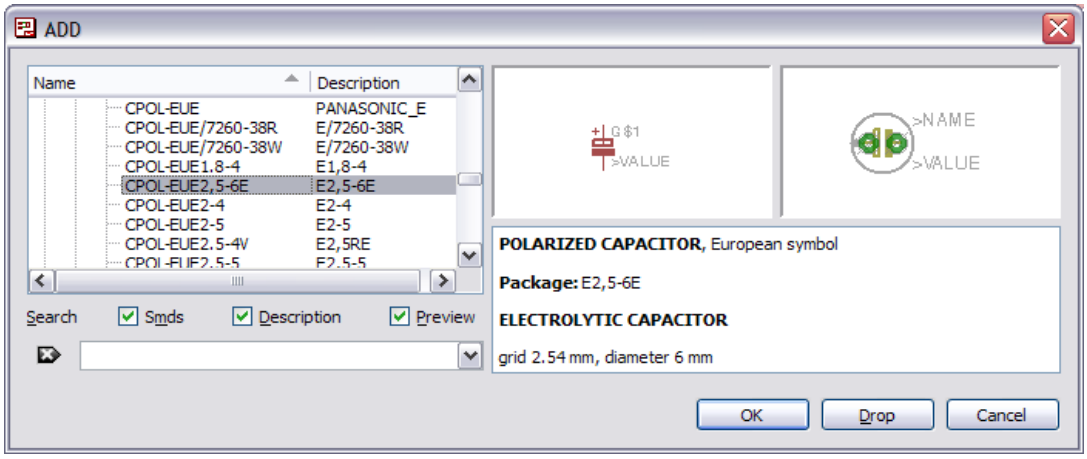


Figura 9.1.3

- Librerías adecuadas a fabricantes concretos

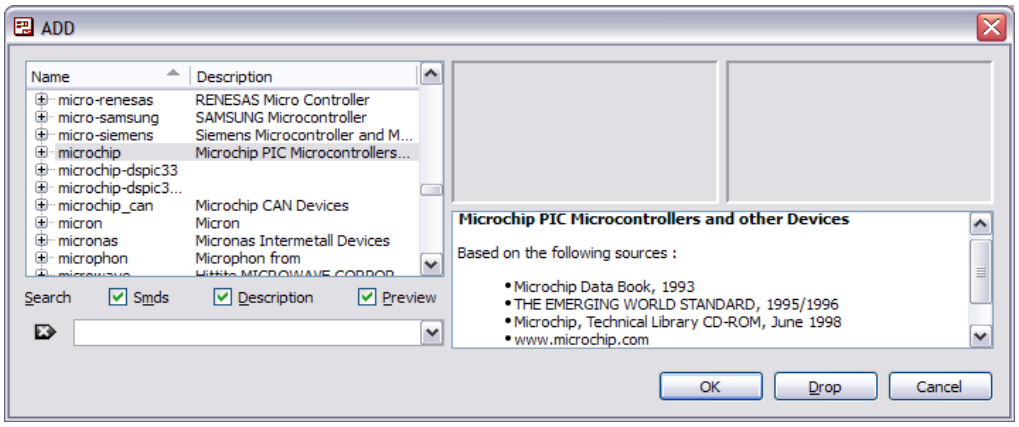


Figura 9.1.4

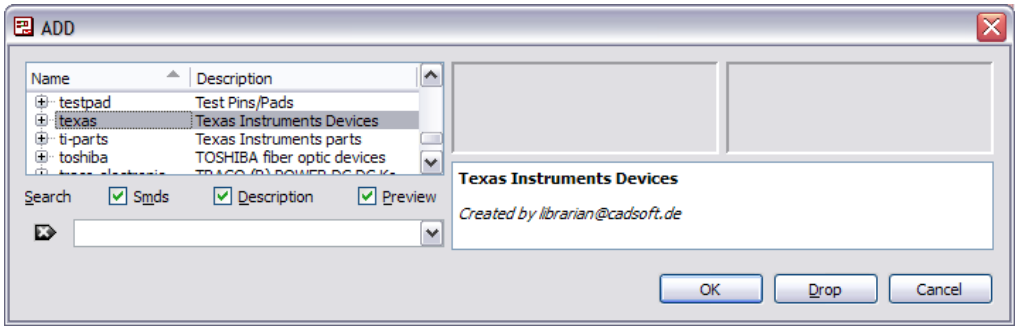


Figura 9.1.5

Una vez colocados todos los componentes a utilizar se deben unir los pines adecuadamente para que se produzcan las conexiones físicas que pueden ser comprobadas en el archivo .brd, todo ello según los *datasheet* de los fabricantes para obtener la funcionalidad deseada.

El aspecto final del esquemático es:

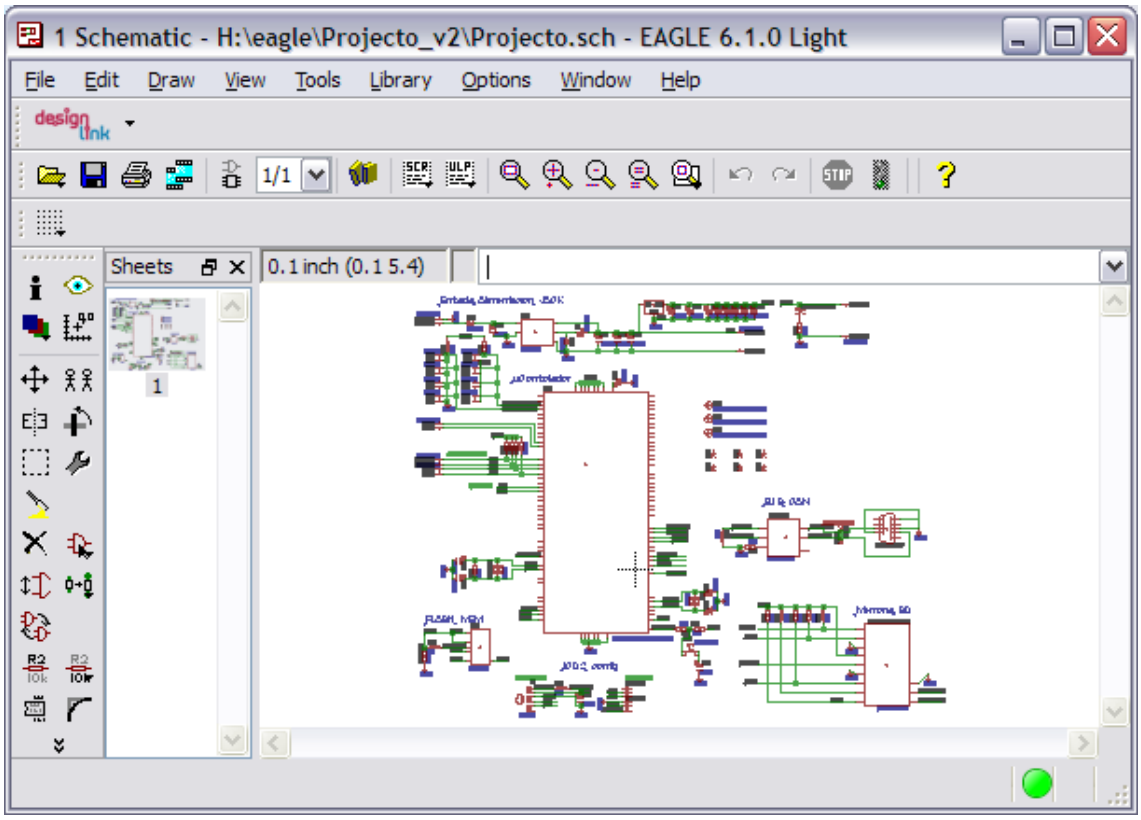


Figura 9.1.6

El esquema está conectado tanto mediante líneas que unen directamente unos pines con otros como con etiquetas para evitar la saturación de líneas cruzadas.

En las siguientes páginas se procederá a analizar individualmente cada bloque para poder clarificar tanto su lectura como su comprensión.

9.2. ALIMENTACIÓN

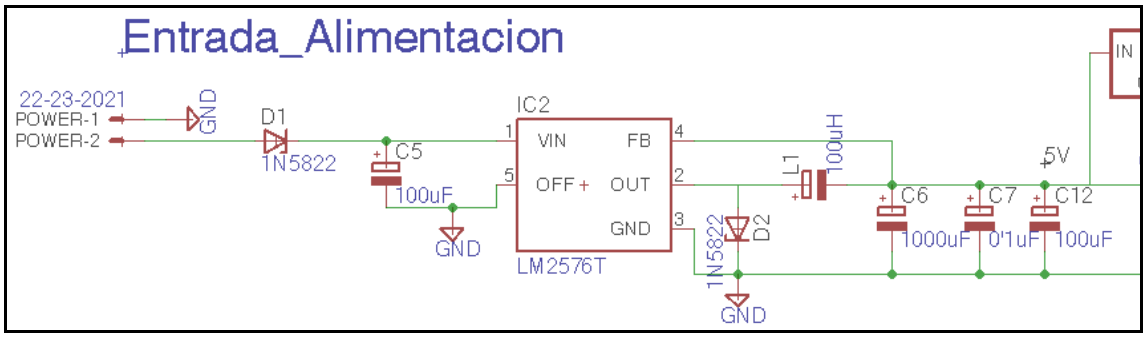


Figura 9.2.1

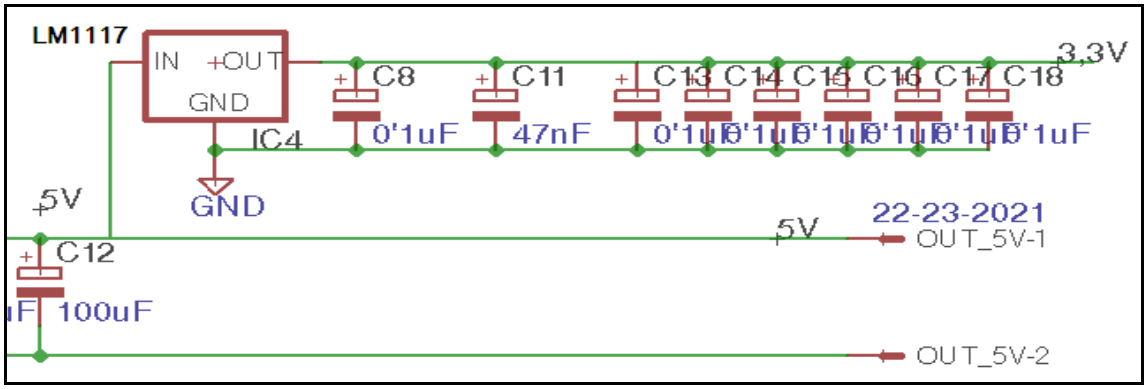


Figura 9.2.2

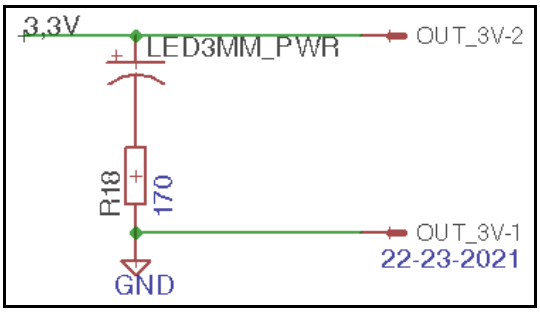


Figura 9.2.3

Obsérvese que hay dos etapas de reducción y estabilización para alimentar la tarjeta:

- La primera etapa obtiene 5V desde 12V mediante el LM2576T
- La segunda obtiene 3,3V desde 5V mediante el LM1117

También se añade un LED indicador de alimentación a la salida para comprobar que tenemos 3,3V y anunciar la alimentación de la tarjeta.

9.3. MEMORIA FLASH

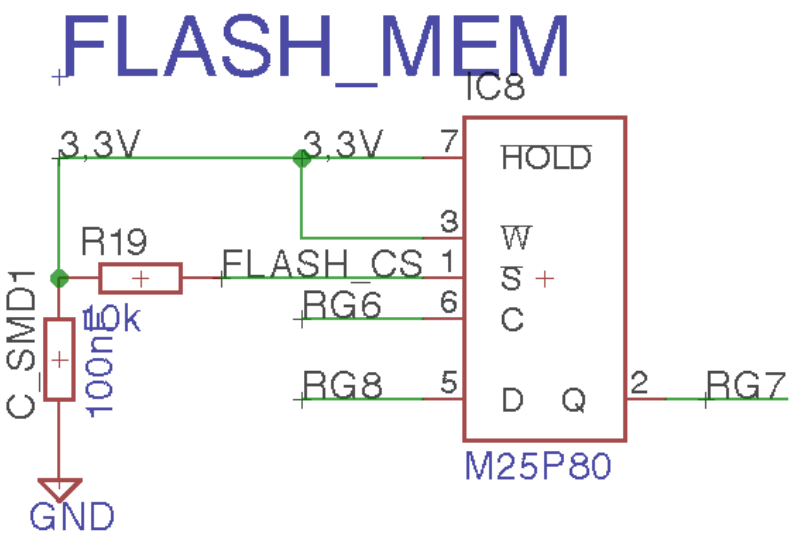


Figura 9.3.1

La memoria flash que permite el almacenamiento de datos (tanto de programa como los leídos por el sensor), lleva las siguientes conexiones con el ucontrolador:

Memoria Flash	Microcontrolador
1 – Chip Select	**9** – AN19/T5CK/T8CK/RC4
2 – Serial Data Output	**11** – RG7/CN9/SDI2
3 – Write Protect	**2** – Vcc
4 – Vss (Ground)	**15** – GND
5 – Serial Data Input	**12** – CN10/SDO2/RG8
6 – Serial Clock	**10** – RG6/CN8/SCK2
7 – Hold	**2** – Vcc
8 – Vcc (Supply Voltage)	**2** – Vcc

Para facilitar el control de la memoria flash y tenerla constantemente seleccionada (ya que en ese bus sólo esta este elemento) se añade el conjunto RC de la figura.

9.4. PROGRAMACIÓN

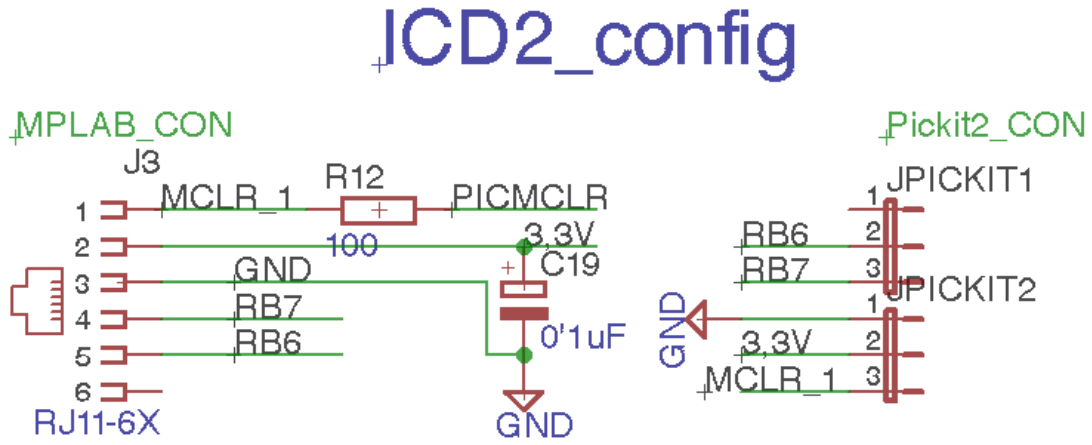


Figura 9.4.1

Como anteriormente se mencionaba, la programación de la tarjeta se podrá realizar mediante dos dispositivos distintos debido a la ambigüedad proporcionada en el diseño. Ambas configuraciones van conectadas al reset global (MRST ó MCLR) dado en la tarjeta.

MPLAB CONFIG	PICKIT2 CONFIG*	MICROCONTROLADOR
1 – MRST	1 – MRST	13 – Master Clear (MRST)
2 – Vcc	2 – Vcc	2 – Vcc
3 – GND	3 – GND	15 – GND
4 – RB7	4 – RB7	27 – RB7/EMUD1/PGD1
5 – RB6	5 – RB6	26 – RB6/EMUC1/PGC1
6 – Not Conected	6 – Not Conected	---

NOTA: (PicKit2 Config*) El esquemático está invertido ya que el conector de este dispositivo es reversible y no depende del orden.

9.5. MEMORIA SD

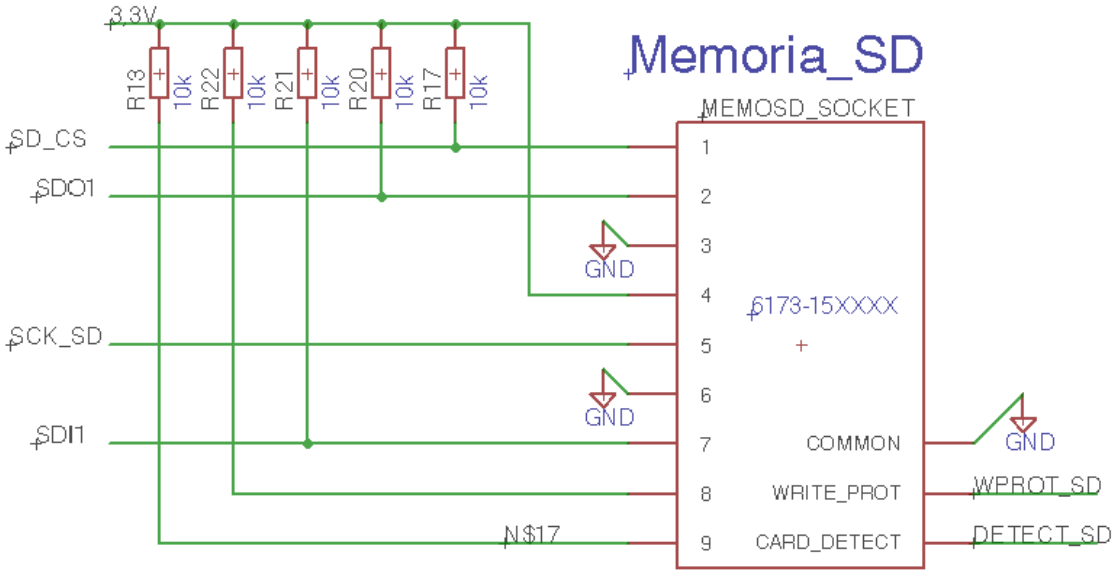


Figura 9.5.1

La tarjeta SD necesita del encapsulado adecuado que irá encajado en una ranura que lo contendrá en la tarjeta para su uso, ya que si se soldase directamente a la tarjeta no se podría extraer y la finalidad de este elemento se perdería.

Sus conexiones son:

Tarjeta SD	Microcontrolador
1 – Chip Select	56 – RG3/SDA1
2 – Serial Data Output	53 – SDO1/RF8
3 – GND	15 – GND
4 – Vcc	2 – Vcc
5 – Serial Clock	55 – SCK1/INT0/RF6
6 – GND	15 – GND
7 – Serial Data Input	54 – SDI1/RF7
8 – Vcc	2 – Vcc
9 – Vcc	2 – Vcc

9.6. BUS CAN

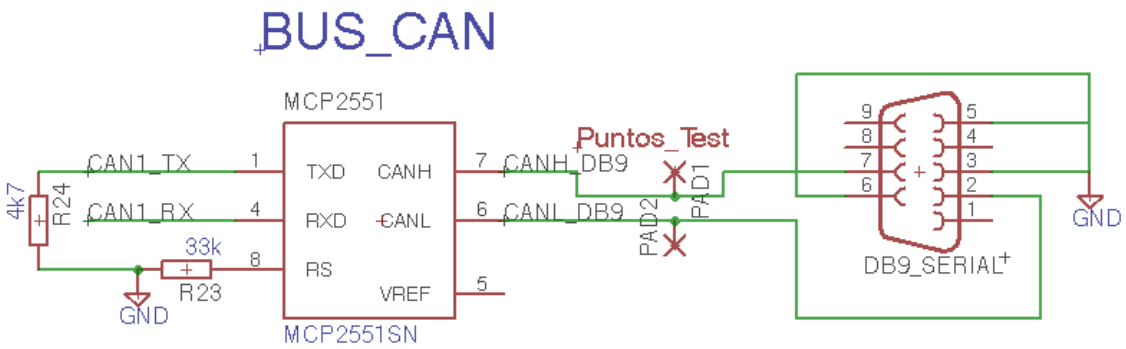


Figura 9.6.1

El bus CAN está gestionado por el microcontrolador, pero como ya fue anteriormente comentado, requiere del MCP2551 para la adaptación de niveles y de un conector adecuado. En este caso el conector DB9 es un conector para la comunicación serie RS232.

La conexión de sus pines es:

Bus CAN	Microcontrolador	RS-232
1 – TXD	88 – C1TX/R1	---
2 – Vss	15 – GND	---
3 – V_{DD}	**Vcc [5V]**	
4 – RXD	87 – C1RX/RF0	---
5 – V_{REF}	---	---
6 – CANL	---	2 – Pin2
7 – CANH	---	7 – Pin7
8 – Rs	15 – GND	---

9.7. µCONTROLADOR

9.7.1. SENSORES

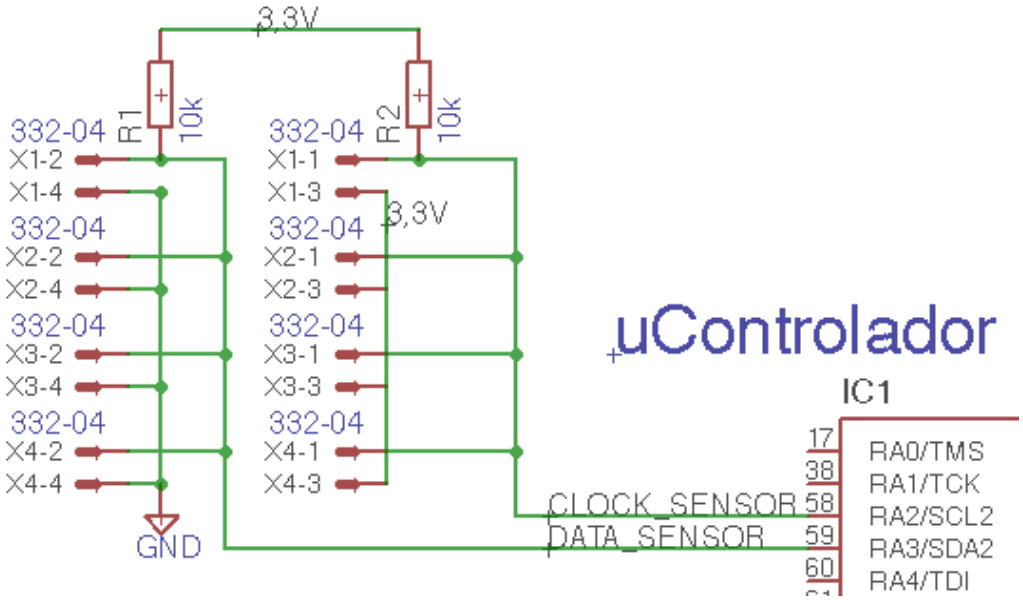


Figura 9.7.1.1

Como ya se comentó anteriormente, los sensores tienen cuatro pines de conexión cada uno: compartiendo dos de ellos de forman común en el bus I^2C y los otros dos como GND y Vcc.

Ya que los sensores no van directamente acoplados en la tarjeta, se han añadido unos conectores de 4 pines cada uno para llevar de forma individualizada a cada sensor las conexiones.

Sensor	Microcontrolador
1 – Serial Clock/Vz	58 – RA2/SCL2
2 – Serial Data/PWM	59 – RA3/SDA2
3 – V_{DD}	2 – Vcc
4 – Vss	15 – GND

9.7.2. INPUT CAPTURE

Figura 9.7.2.1

Simplemente se añade un biconector a estos módulos para futuras implementaciones.

9.7.3. ADCs

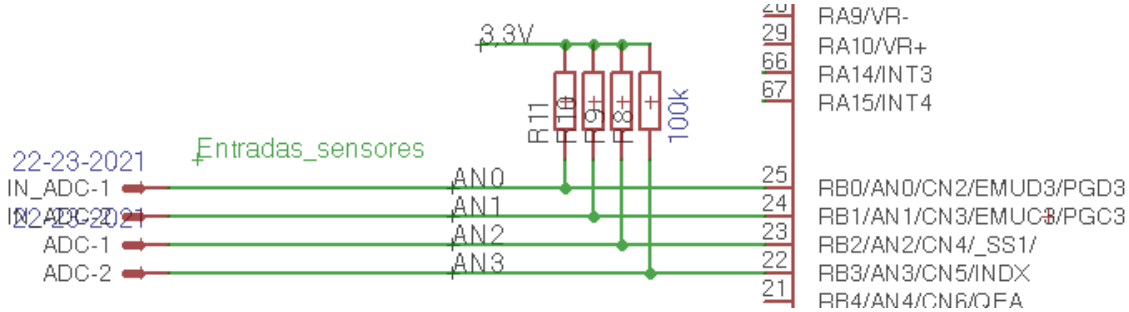


Figura 9.7.3.1

Simplemente se añaden unos conectores de entrada/salida para futuras implementaciones.

9.7.4. RESET MAESTRO

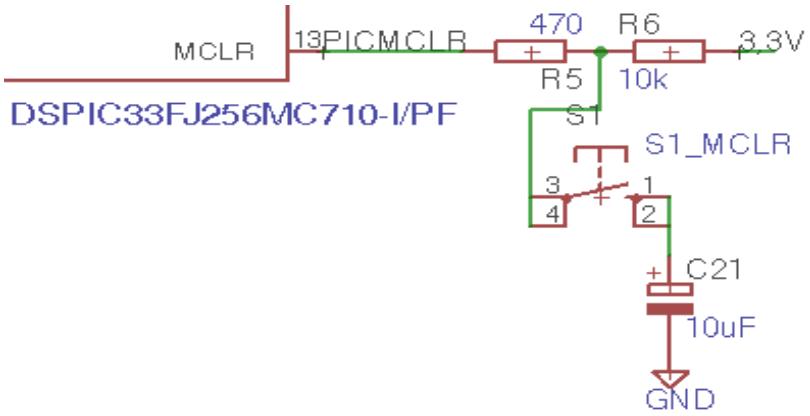


Figura 9.7.4.1

Simplemente se añade un pulsador para poder realizar el reset asíncrono de la tarjeta de forma manual.

9.7.5. OSCILADOR PRIMARIO

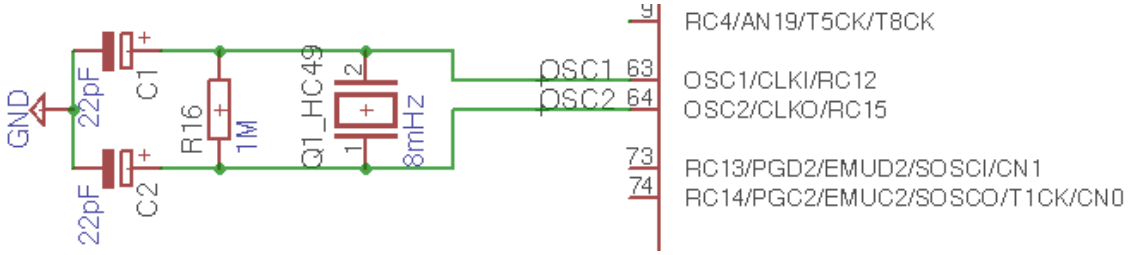


Figura 9.7.5.1

Esta sencilla configuración permite el uso del oscilador primario al configurar adecuadamente los registros que a este elemento hacen referencia.

9.7.6. OSCILADOR SECUNDARIO

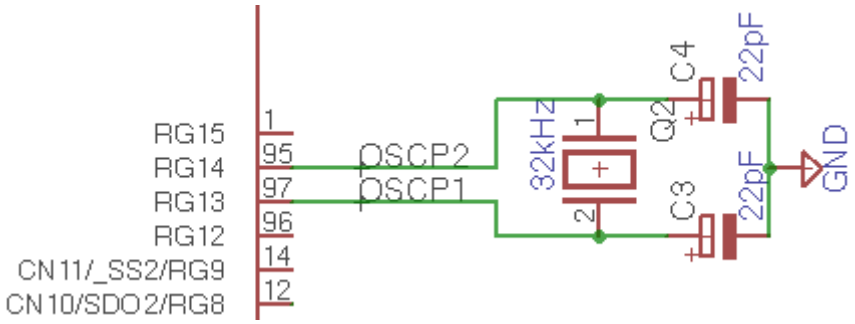


Figura 9.7.6.1

Como anteriomente se comenaba, para tener la disponibilidad del oscilador secundario es necesario añadir estos tres elementos.

9.7.7. VCC Y GND

Y por último, sólo queda indicar los pines del micro que necesitan ir conectados tanto a la alimentación (Vcc) como a masa (GND).

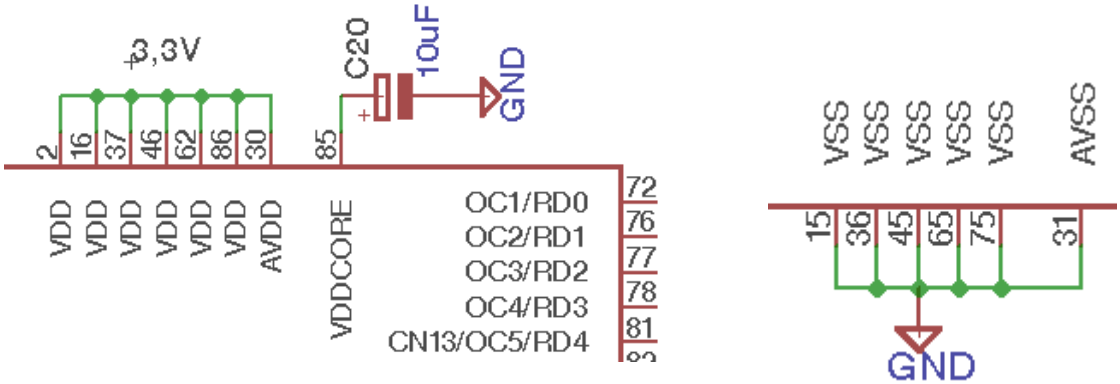


Figura 9.7.7.1

9.8. TARJETA DE CIRCUITO IMPRESO

Una vez realizado todo el conexionado en el ".SCH" se debe continuar con la siguiente fase del diseño: El archivo ".BRD".
Tras colocarse los componentes a lo largo y ancho de las dimensiones de la tarjeta, se deberá realizar el ruteado para que las conexiones físicas se materialicen de líneas de conexión a pistas de conexión. Sin duda, este es un punto importante del diseño, ya que realizar un ruteado completo manualmente puede ser arduo y generar un ruteado automático depende de la configuración minuciosa que se haga del autorouter.

Sin embargo, finalmente se realiza un autoruteo para posteriormente corregir manualmente las imperfecciones y/o realizar las rutas que hayan quedado pendientes.

Autoruteo

- Por defecto, en Eagle se establece el ancho de pista como el mínimo de los PADs que haya en nuestro diseño: 10 mils (ó 0,22 milímetros) en nuestro caso debido al microcontrolador de encapsulado TQFP. Este parámetro se mantiene por defecto.
- La distancia entra PADs SMD y pistas se mantiene en 10 mils.
- La distancia entre PADs de inserción y pistas se establece a 24 mils.
- La distancia entre pistas se mantiene a 10 mils, pero posteriormente se hace un repaso completo para separar a 24 mils las que sea posibles distanciar.
- El ancho de los PADs se mantiene en sus valores por defecto.

Tras guardar la configuración del autorouter se tendrán dos opciones, rutear directamente con Eagle o exportar el diseño (es decir, lo pads, su tamaño, sus interconexiones y la configuración para el ruteamiento deseada) a otro ruteador más potente.
Finalmente se opta por exportar el archivo (con extensión ".DSN") a un programa java denominado FreeRouting que realiza un ruteado completo, el cual añade las vias necesarias y después optimiza el conexionado para minimizar el número de vías y de pistas en el dibujo.

Después sólo queda importar el archivo de extensión ".SCR" a Eagle (se dibujarán las pistas automáticamente y se podrán realizar las últimas mejoras de forma manual). Obtenemos una tarjeta con el siguiente aspecto:

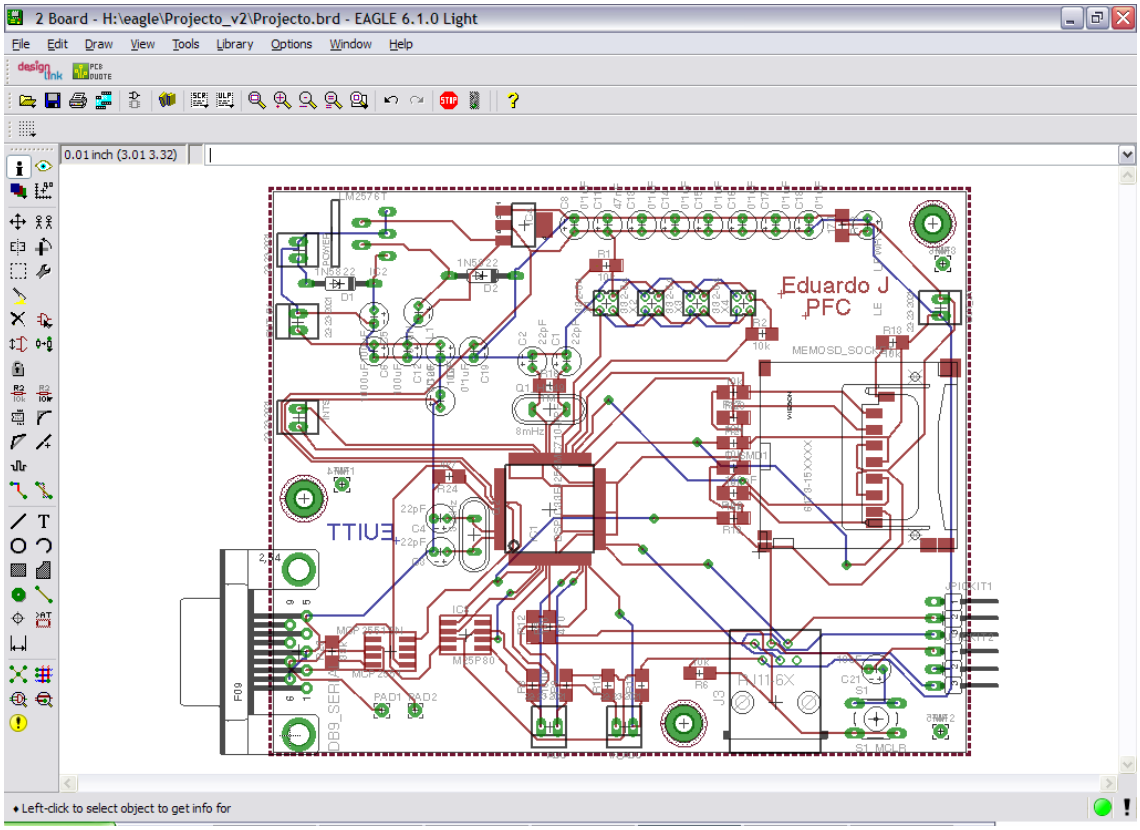


Figura 9.8.1

Donde la zona más conflictiva, la del microcontrolador, se debe repasar a mano concienzudamente para evitar posteriores problemas al imprimir el fotolito y al obtener la tarjeta físicamente en la placa de cobre:

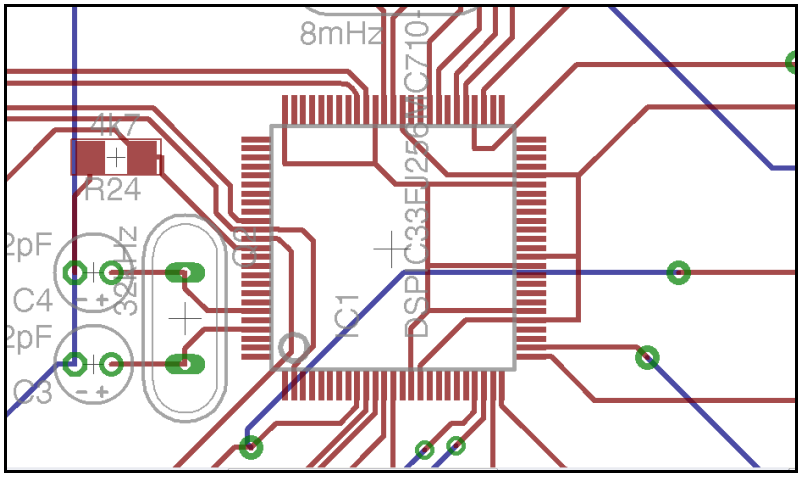


Figura 9.8.2

Imagen de la cara top:

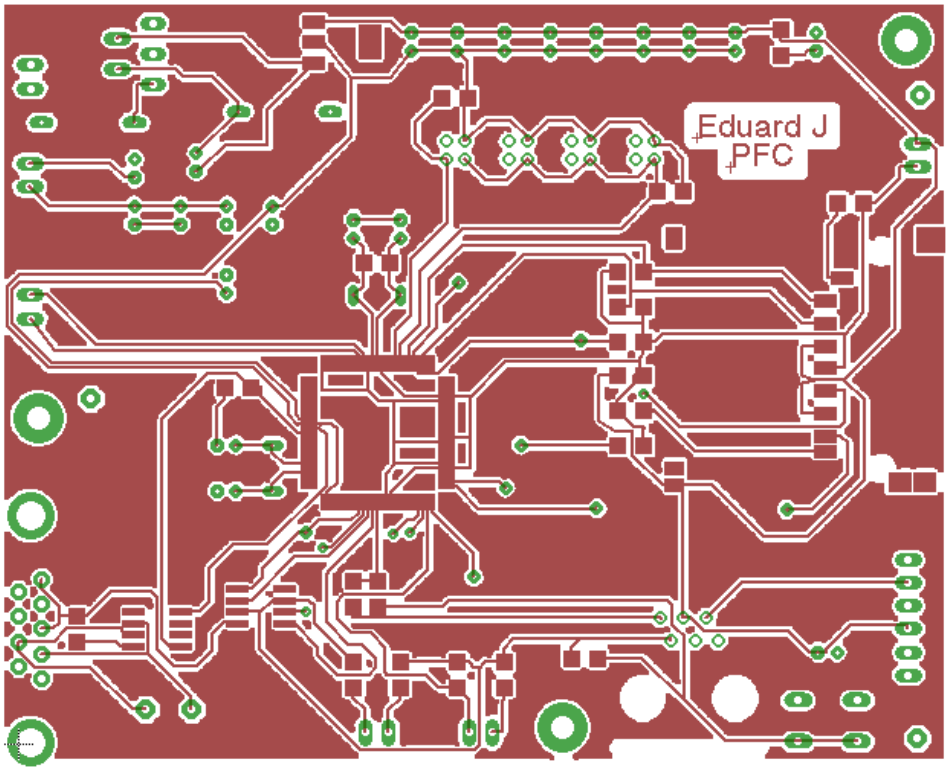


Figura 9.8.3

Imagen de la cara bottom:

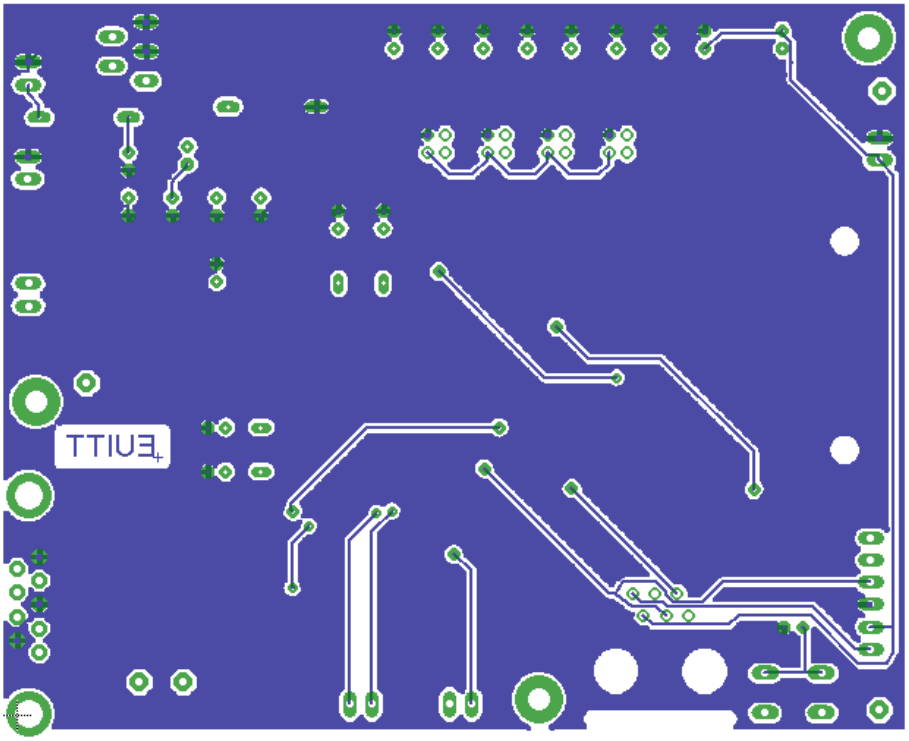


Figura 9.8.4

Colocación de los componentes:

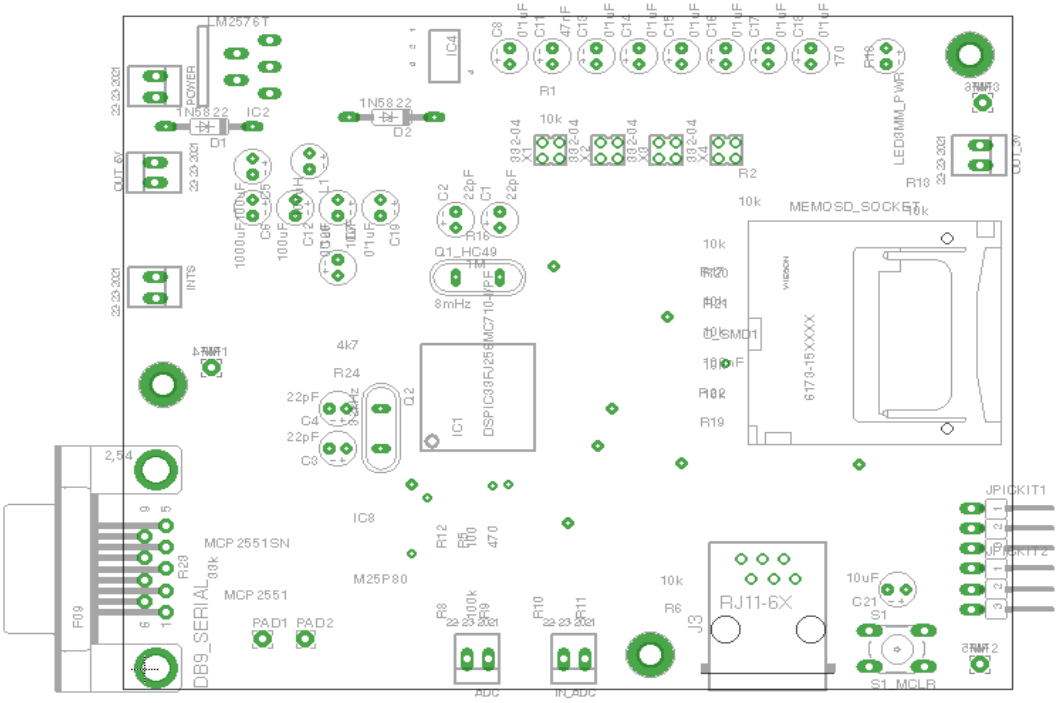


Figura 9.8.5

9.8.1. GENERACIÓN DEL FOTOLITO EN EAGLE

Una vez que se tiene la placa planteada y dispuesta para imprimir su fotolito, se puede generar el fotolito mediante dos métodos:

- Se puede generar **directamente el imprimible en formato .PDF** siguiendo los siguientes pasos, los cuales fueron los realizados para la implementación física:
  1) En la barra de opciones superior debe asegurar el uso del fondo blanco: Options → Use Interface… → Layout → Background White
  2) Después: View → Display/Hide layers… → None
  3) Ahora se marcan: top, pads y vias (coloreadas en negro).
  4) Finalmente: File → Print… → Printer → Print to file (PDF)

  Repetir este proceso para la cara bottom desde el punto 2.

NOTA: Se aconseja utilizar un escalado de 1.01 en este proyecto al crear el .PDF, así como una impresora láser de alta resolución para obtener un fotolito más fiable y cómodo para el insolado del cobre.

- Ó se pueden **generar los GERBERS** de la tarjeta mediante el generador de gerbers de Eagle. Esto permite la opción de presupuestar e implementar profesionalmente el diseño realizado. Para ello se debe seguir el proceso mostrado a continuación:
  1) En la barra de opciones superior: File → CAM Processor
  2) En la nueva ventana: File → Open… → Job → Archivo "gerb274x.cam"
  3) Se abrirán una serie de pestañas con unos valores por defecto, pero las modificaremos para obtener los archivos deseados que se generarán, tras dicha verificación, pulsando "Process Job" (el cual genera todas las secciones incluidas en la parte superior).

En este proyecto las secciones incluidas han sido los siguientes:

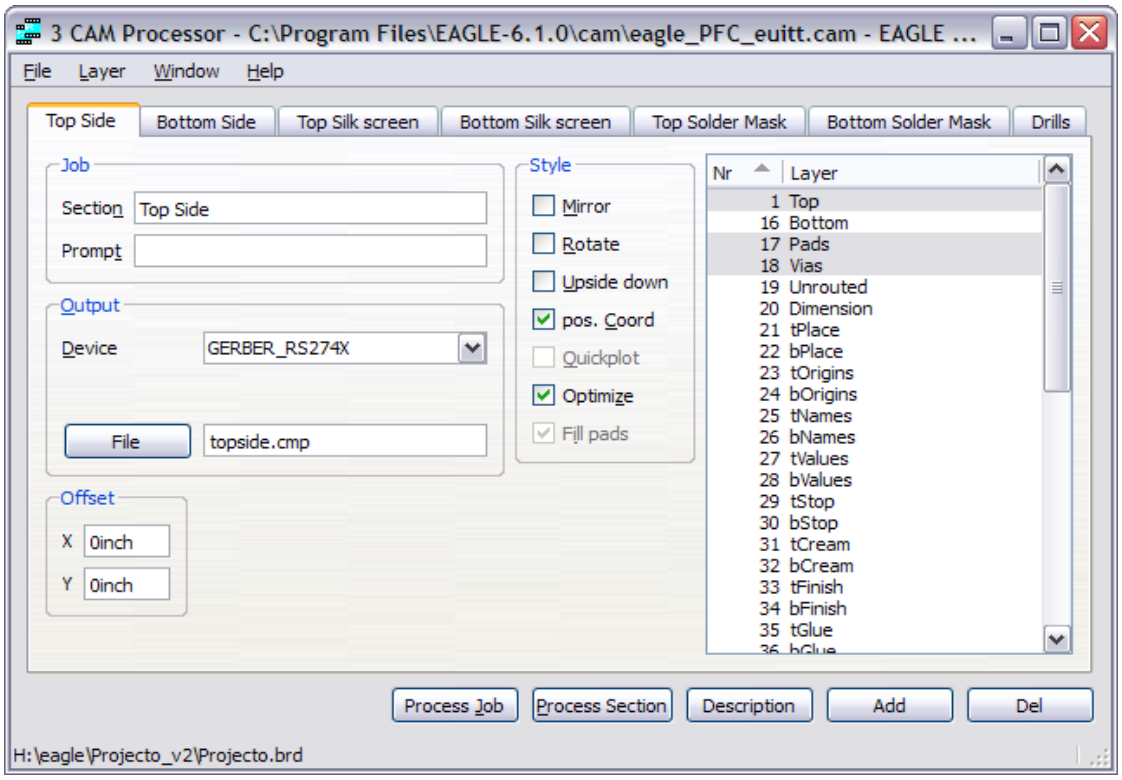


Figura 9.8.1.1

Donde los valores que se deben incluir en cada pestaña (o sección) son:

File Extension	Device	Layers	Description
cmp	GERBER_RS274X	Top, Pads, Vias	Top layer copper
sol	GERBER_RS274X	Bottom, Pads, Vias	Bottom layer copper
plc	GERBER_RS274X	Dimension, tPlace, tNames	Top silk screen
pls	GERBER_RS274X	Dimension, bPlace, bNames	Bottom silk screen
stc	GERBER_RS274X	tStop	Top solder mask
sts	GERBER_RS274X	bStop	Bottom solder mask
drd	EXCELLON	Drills, Holes	Drills

En las siguientes figuras (Figuras 9.8.1.2 y 9.8.1.3) se ven los fotolitos impresos en papel transparente desde formato .PDF:

Cara TOP:

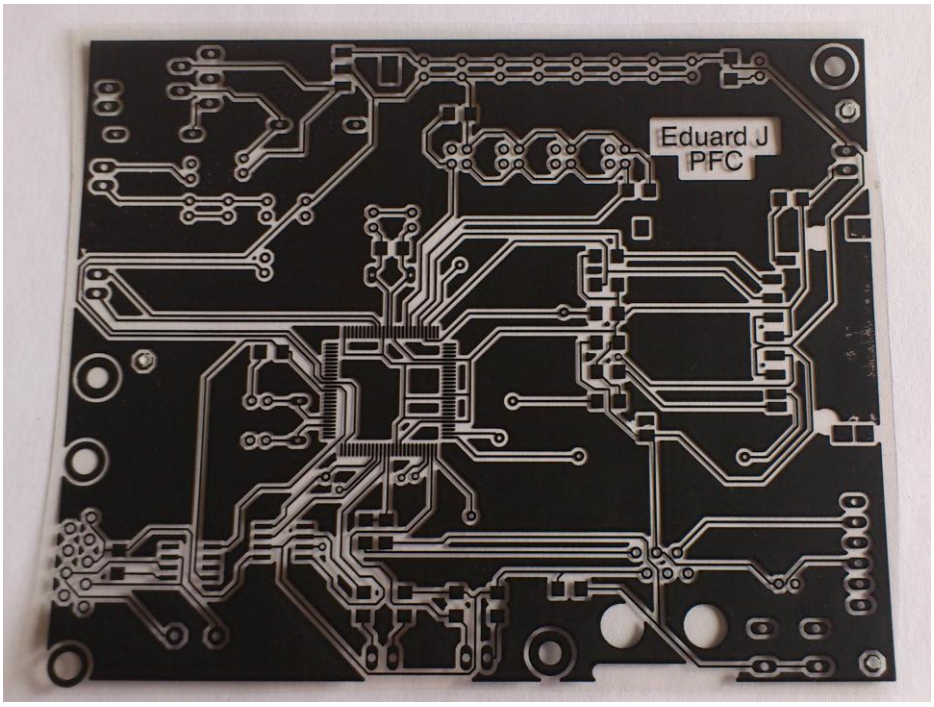


Figura 9.8.1.2

Cara BOTTOM:

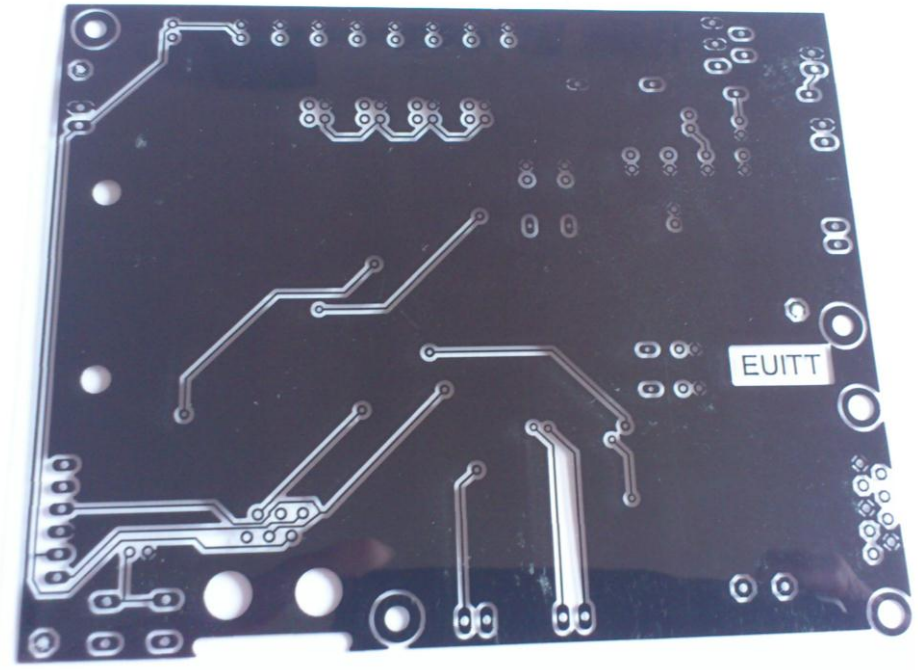


Figura 9.8.1.3

Y por tanto la placa con el esquema debe presentar el siguiente aspecto:

Cara TOP:

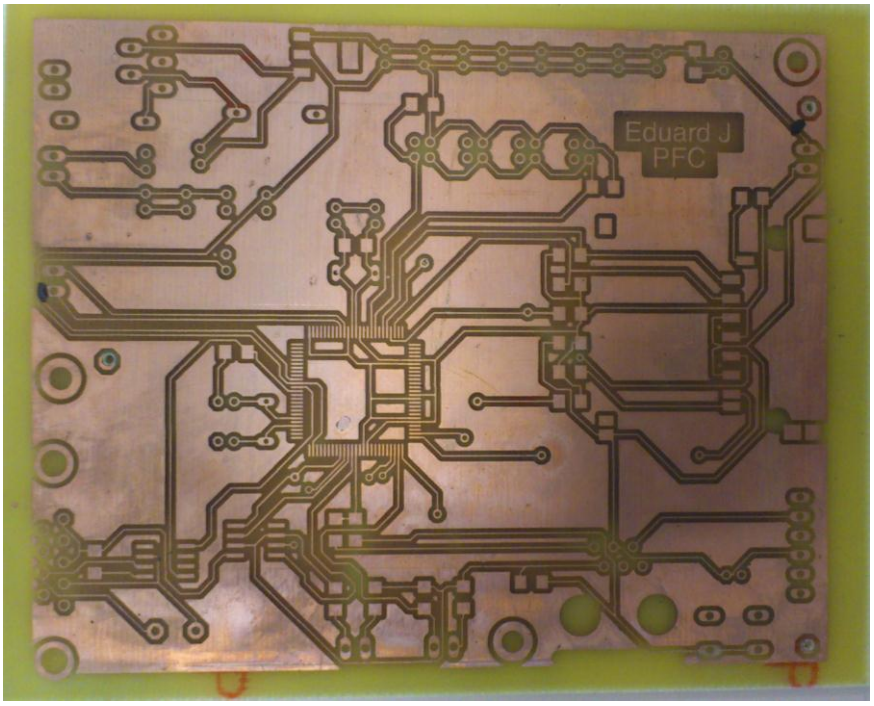


Figura 9.8.1.4

Cara BOTTOM:

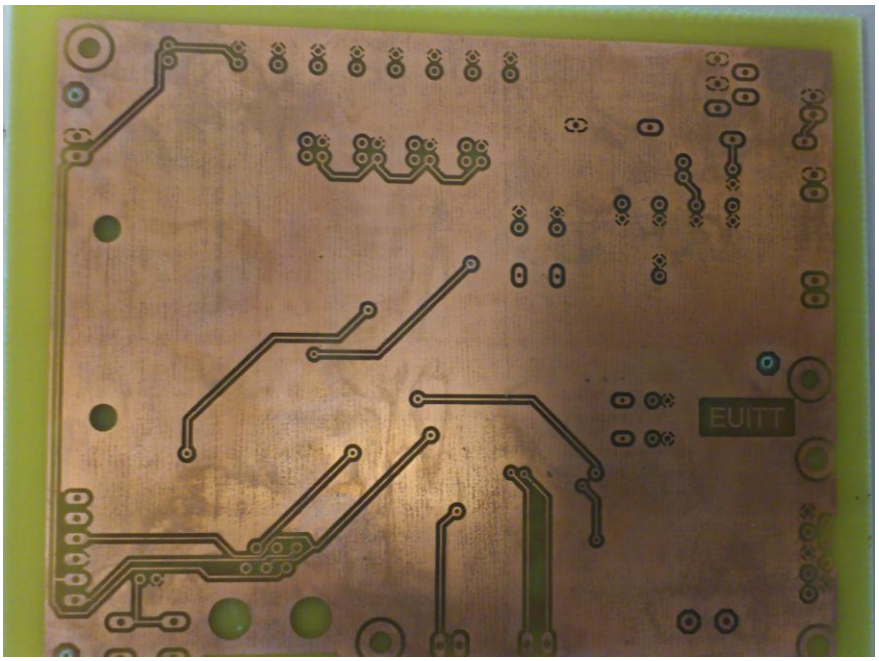


Figura 9.8.1.5

Consejos para realizar la tarjeta

1. Para encuadrar las dos caras se aconseja taladrar la placa en tres o cuatro sitios distintos y ubicados hacia los extremos con uno de los dos fotolitos sobre la placa y ajustar después el segundo también taladrado en los mismos pads (o marcas fiduciales auxiliares) para encuadrar ambas caras fácilmente.
2. Debido al reducido tamaño tanto de algunos pads como de las pistas, es aconsejable utilizar al realizar la placa:
    - Insolado de 120 segundos con una insoladora de 160W.
    - Un revelado suficiente como para distinguir al detalle entre pads TQFP.
    - Una mezcla suave en el proceso de atacado (cloruro férrico disuelto en agua en proporción del 50% cada uno).

Como se muestra en la figura 9.8.1.6, es conveniente revisar minuciosamente las partes más problemáticas del esquema una vez se ha realizado la tarjeta:

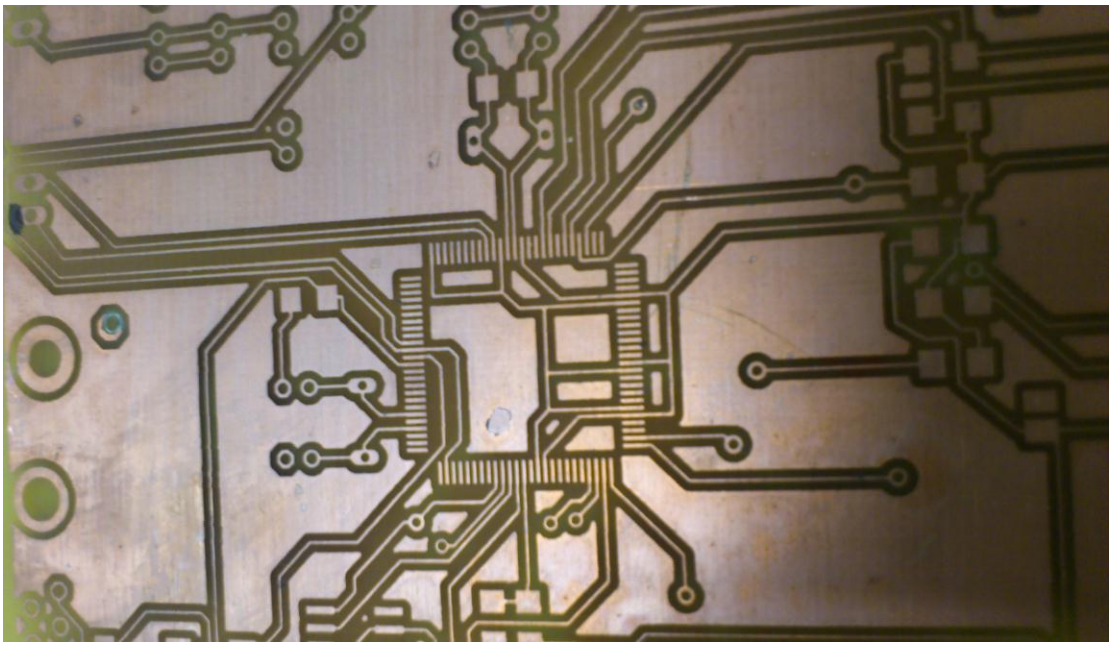


Figura 9.8.1.6

9.8.2. SOLDADURA

Una vez que se limpiado con acetato (o son alcohol) el cobre y se ha comprobado la continuidad de las pistas y subsanado las posibles imperfecciones que hayan aparecido, es hora de soldar.

Para ello, se ha de empezar por las partes más complicadas y de difícil acceso ya que se van a ir complicando a medida que añadamos los componentes a la tarjeta. Esto conlleva a comenzar siempre por los componentes SMD y después pasar a los de inserción.

Fotografías de la soldadura del microcontrolador en horno:

Figura 9.8.2.1

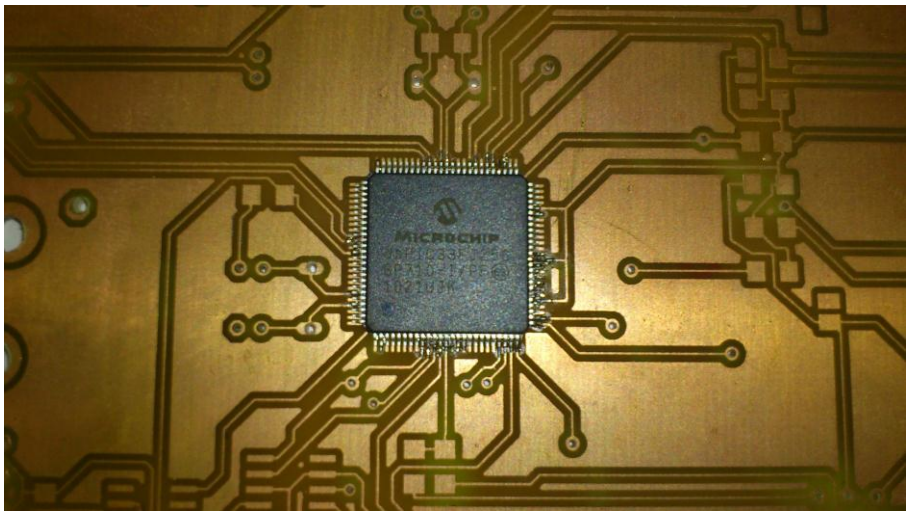


Figura 9.8.2.2

Fotografía de la soldadura completa de los componentes en la PCB (CARA TOP):

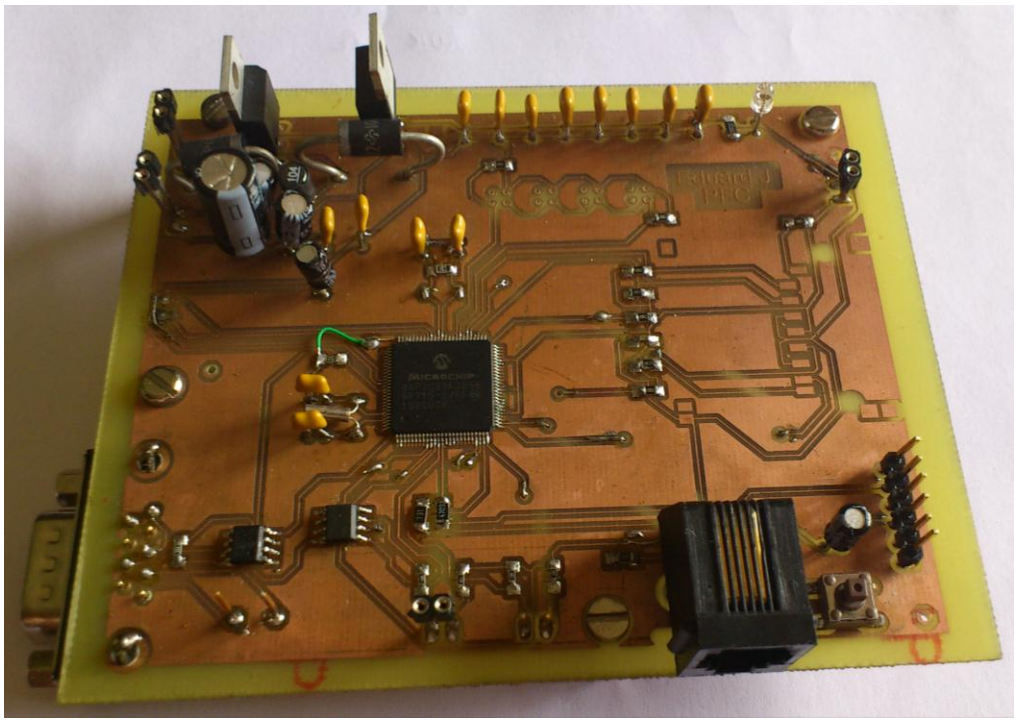


Figura 9.8.2.3

Fotografía de la soldadura completa de los componentes en la PCB (CARA BOTTOM):

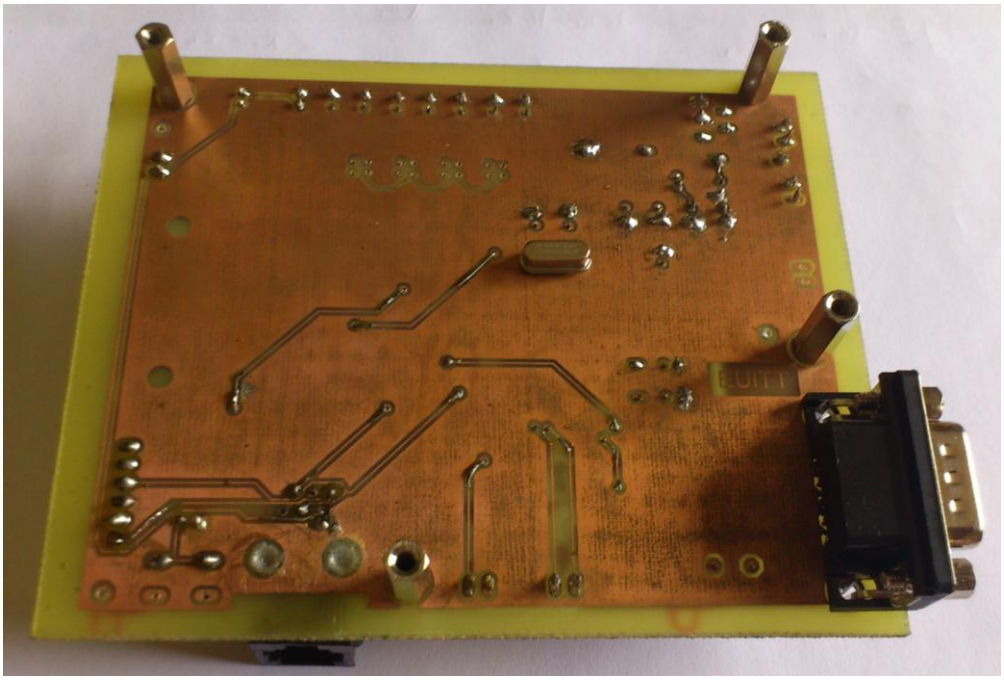


Figura 9.8.2.4

10. DESARROLLO SOFTWARE

Respecto a este apartado software, todo el código y su depuración se ha llevado a cabo mediante el entorno de desarrollo facilitado por Microchip: MPLAB IDE v8.56.

10.1. ¿QUÉ ES MPLAB IDE?

MPLAB IDE (Integrated Development Enviroment) es un conjunto de herramientas libres e integradas para el desarrollo de aplicaciones embebidas que utilizan microcontroladores PIC y dsPIC. MPLAB IDE se ejecuta como una aplicación de 32 bits de Microsoft Windows, la cual es fácil de usar e incluye una serie de componentes de software libre para el rápido desarrollo de aplicaciones y su depuración. MPLAB también sirve como una interfaz gráfica de usuario unificada para Microchip y el software de terceros, incluyendo herramientas de desarrollo hardware. Moverse entre las herramientas es muy fácil, y la actualización del simulador de software libre para depuración de hardware y herramientas de programación se realiza en un instante, porque MPLAB tiene la misma interfaz de usuario para todas las herramientas.
El compilador y los linkadores que utiliza permiten la programación orientada a un único microcontrolador ó generalizar para un extenso rango de microcontroladores.

Para comenzar un proyecto se debe acceder a Project → Project Wizard… y seguir los pasos casi automáticos hasta finalizar el proceso.
Préstese especial atención a la selección del microcontrolador (Figura 10.1), ya que es imprescindible localizar el modelo exacto que se va a programar para que el entorno pueda identificarlo una vez se interconecte el ordenador con el microcontrolador. Aunque también puede modificarse a posteriori el dispositivo.

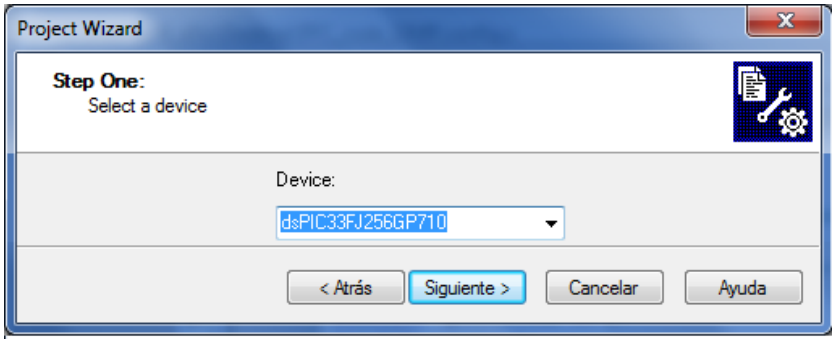


Figura 10.1

En la figura 10.2 se muestra una de las ventajas aportadas por el entorno, ya que al seleccionar el dispositivo permite ver las herramientas que soporta cada microcontrolador. Para este proyecto se ha utilizado el µC dspic33fj256gp710:

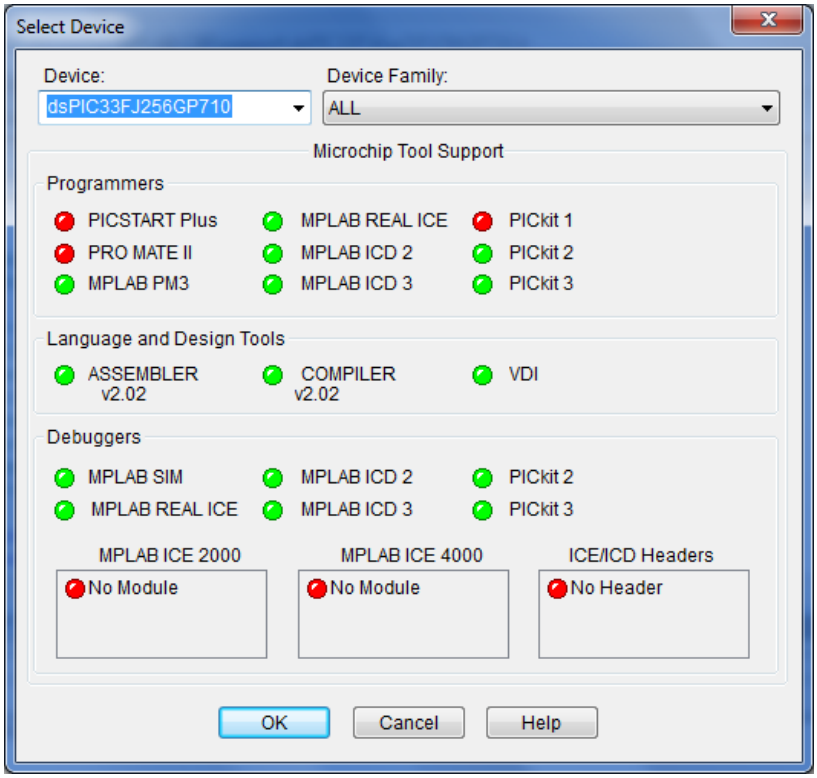


Figura 10.2

Apréciese el resultado final en la figura 10.3:

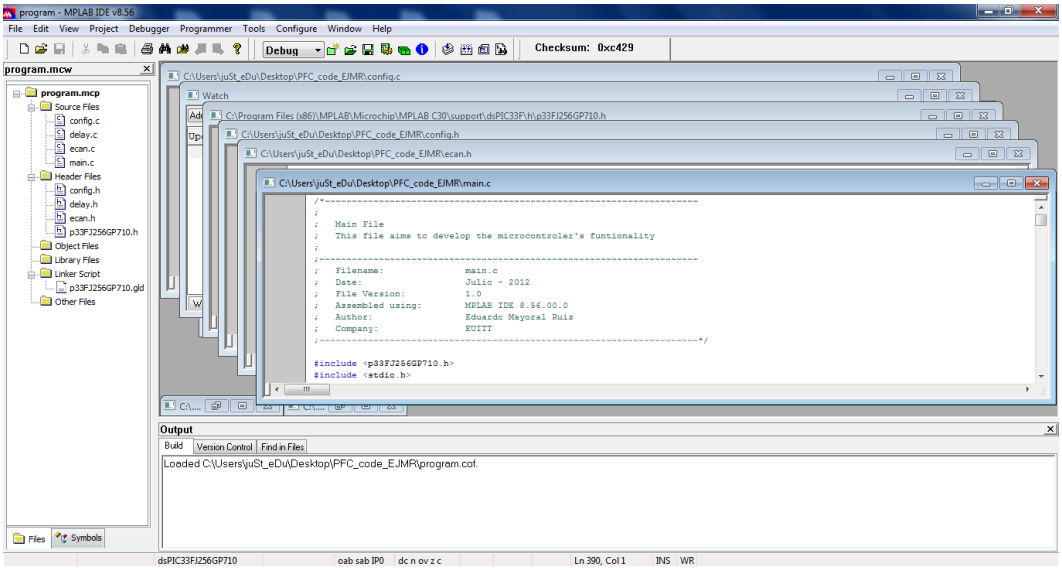


Figura 10.3

10.2. DESARROLLANDO EL CÓDIGO

En este apartado se van a explicar los puntos claves que surgieron al construir la aplicación software y requieren de configuración en frecuencia:

10.2.1. Oscilador del µControlador

Para configurar el sistema a la frecuencia de oscilación deseada, se ha decidido utilizar el cristal oscilador que se introdujo en la tarjeta de 8MHz. Para ello se recurre al *datasheet* del µC y a la información adicional y específica sobre osciladores en dsPics facilitada por Microchip para consultar las fórmulas que establecen las relaciones entre los registros a configurar y la frecuencia resultante:

$$FCY = \frac{FOSC}{2} \quad \text{y} \quad FOSC = FIN \cdot \left(\frac{M}{N1 \cdot N2}\right)$$

Igualando se obtiene:

$$Fcy = \frac{8MHz}{2}\left(\frac{M}{N1*N2}\right)$$

Donde M = 40, N1 = 2 y N2 = 2 por lo que resulta **FCY = 80MHz → 40MIPS**

En los registros asociados al oscilador se establecerán los siguientes valores:

```
PLLFBD = 38;                // M = (40 – 2)
CLKDIVbits.PLLPOST = 0;     // N1 = (2 – 2)
CLKDIVbits.PLLPRE = 0;      // N2 = (2 – 2)
```

El diagrama de la figura 10.2.1.1 ofrece una visión de este proceso:

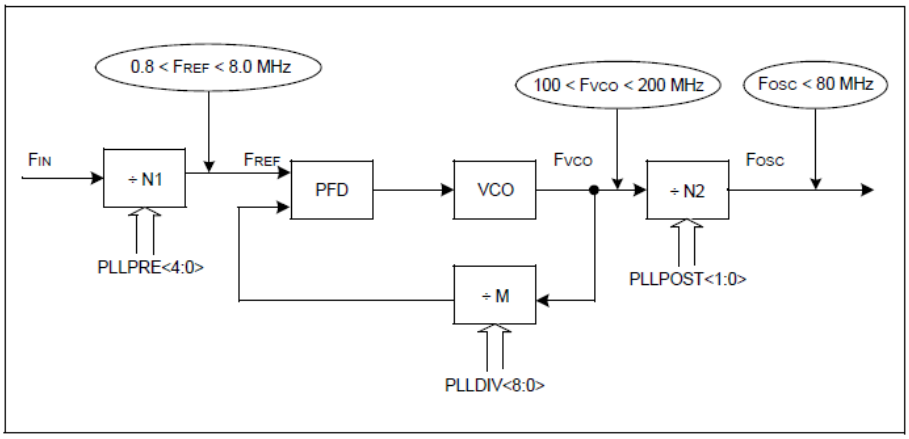


Figura 10.2.1.1

10.2.2. I2C y Sensores

Para este apartado se van a tratar los siguientes aspectos:

- Configuración del puerto I2C(2)
- Función de lectura
- Función de escritura

Configuración del puerto I2C(2)

Debido a que el sensor MLX90614 sólo soporta velocidades comprendidas entre 10KHz y 100KHz, se debe establecer un Baud Rate adecuado a estas frecuencias en el registro del µC, I2CBRG, acorde a la siguiente relación:

$$I2CBRG = \left[\left(\frac{1}{F_{SCL}} - PGD\right) \cdot F_{CY}\right] - 2$$

En la siguiente tabla se pueden apreciar algunas relaciones obtenidas del catálogo en base a la fórmula anterior:

Required System FSCL	FCY	PGD(1)	I2CxBRG Decimal	I2CxBRG Hexadecimal
100 kHz	40 MHz	130 ns	392.8	0x188
100 kHz	20 MHz	130 ns	195.4	0x0C3
100 kHz	10 MHz	130 ns	96.7	0x060
400 kHz	20 MHz	130 ns	45.4	0x02D
400 kHz	10 MHz	130 ns	21.7	0x015
400 kHz	5 MHz	130 ns	9.85	0x009
1 MHz	10 MHz	130 ns	6.7	0x006

Note 1: The typical value of the Pulse Gobbler Delay (PGD) is 130 ns. Refer to the specific device data sheet for more information.

También se muestran algunas relaciones extra no reflejadas en el *datasheet*:

FSLC [kHz]	PGD [ns]	FCY [MHz]	I2CxBRG_DEC	I2CxBRG_HEX
10	130	40	3992,0	0x0F9E
20	130	40	1992,0	0x07CE
30	130	40	1325,3	0x0533
40	130	40	992,0	0x03E6
50	**130**	**40**	**792,0**	**0x0318**
60	130	40	658,7	0x0299
70	130	40	563,4	0x023A
80	130	40	492,0	0x01F2
90	130	40	436,4	0x01BA
100	130	40	392,0	0x0188

Función de lectura

Debido a que el fabricante diferencia entre una función de lectura con una de escritura, merece la pena destacar detalladamente las diferencias para poder entender y manejar correctamente el protocolo I2C en la comunicación con el sensor MLX90614.

Esta función persigue la lectura repetitiva de las medidas registradas por los sensores para obtener constantemente las temperaturas deseadas. Siempre teniendo en cuenta que un sensor necesita 100 milisegundos para establecer una nueva medida en sus registros.

En la figura 10.2.2.1 se aprecia el protocolo a seguir en la comunicación antes mencionada para la lectura de un dato del sensor:

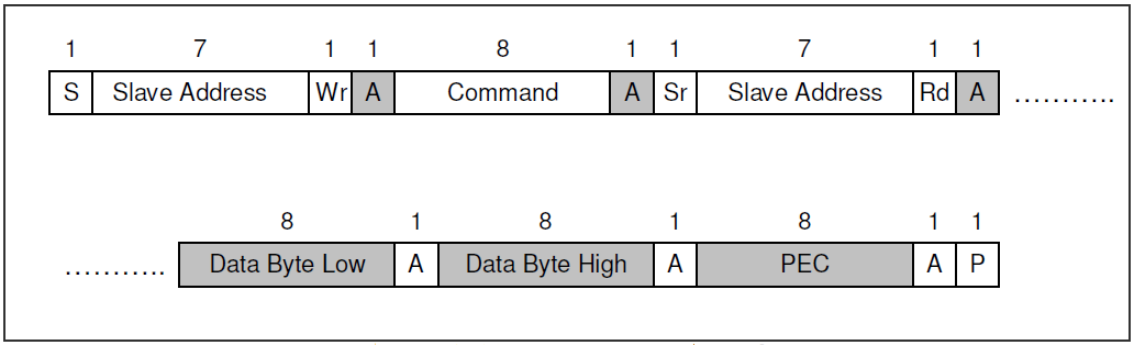


Figura 10.2.2.1

Como se aprecia son necesarios 5 bytes de datos para gestionar esta información:

- **SlaveAddress**: Byte con la dirección del sensor a leer desplazando hacia la izquierda dicho dato para poder añadir el bit de Wr/Rd.
  La dirección por defecto en este modelo de sensor es 0x5A, lo cual implica que desplazando obtenemos:
  [0x5A] 0101 1010 → [0xB4] 1011 0100
- **Command**: Byte con el comando de interactuación con el sensor, que debe cumplir lo siguiente:

Opcode	Command
000x xxxx*	RAM Access
001x xxxx*	EEPROM Access
1111_0000**	Read Flags
1111_1111	Enter SLEEP mode

Figura 10.2.2.2

NOTA: Las equis deben remplazarse por la dirección deseada.

- **DataByteLow**: Byte que nos devolverá el sensor con la parte baja de los datos ubicados en la dirección que se le ha especificado.

- **DataByteHigh**: Byte que nos devolverá el sensor con la parte alta de los datos ubicados en la dirección que se le ha especificado.

- **PEC**: Byte de datos que recoge el checksum de la función siguiendo un CRC-8-CCITT (Cyclic Redundancy Check), es decir, x^8+x^2+x+1.
  En la función de lectura, este acumulador tiene en cuenta para el cálculo del checksum los siguientes bytes:
  SlaveAddress + Command + (SlaveAddress+1) + DBLow + DBHigh

  Se añaden un par de ejemplos ya calculados:
  B4 + 07 + B5 + CF + 3B ➔ PEC = 89
  B4 + 07 + B5 + EC + 3B ➔ PEC = 18
  NOTA: Para calcular este checksum se necesitan 5 bytes.

  O también como ya se vio anteriormente:

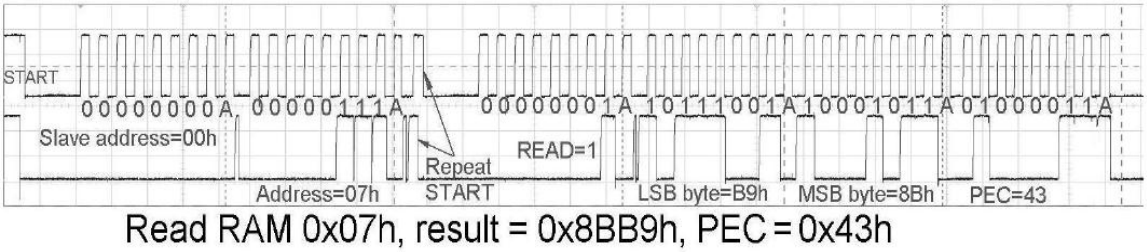


Figura 10.2.2.3

Función de escritura

Téngase en cuenta que en un bus I2C no pueden existir dos dispositivos que respondan a la misma SlaveAddress, por tanto al tener cuatro sensores en esta aplicación, como mínimo se le deberá de modificar la dirección que tienen por defecto estos elementos a tres de ellos, lo cual obliga a desarrollar una función de escritura. De hecho, si en vez de utilizarse la configuración SMBus se utilizase la configuración PWM, debería de desarrollarse la función de escritura para modificar el registro de configuración del sensor porque por defecto viene configurado para SMBus.

La función debe seguir el protocolo mostrado en la figura 10.2.2.4:

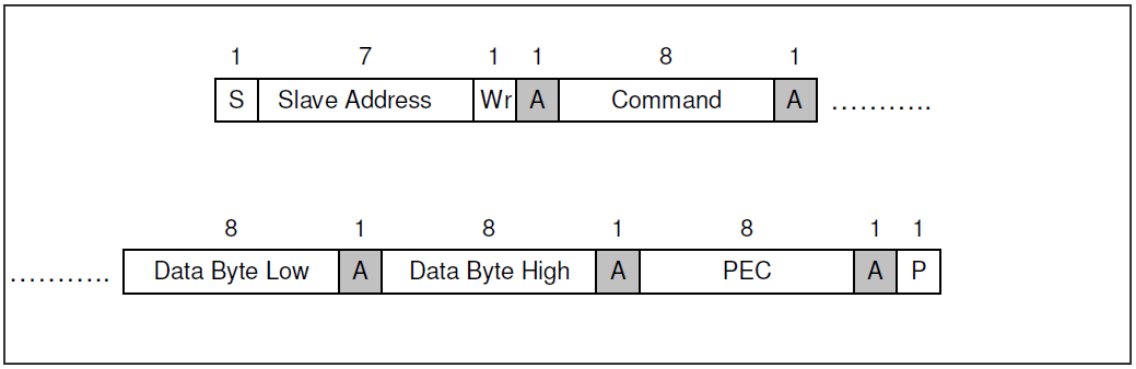


Figura 10.2.2.4

Como se ve en la figura 10.2.2.4 ahora todos los bytes los envía el µC y el sensor simplemente devuelve los ACKs tras cada byte.
La parte más complicada a desarrollar es el envío del PEC. Sin embargo este byte se puede introducir calculándolo con un programa de forma externa y añadiendo el valor resultante al registro que envía los datos de salida al I2C. Pese a esto, finalmente se decidió incorporar al código una serie de funciones que resolviesen esta problemática para, entre otras cosas, hacer una comprobación en la función de lectura para cada medida.

Cálculo del PEC con un ejemplo:
Command + DBLow + DBHigh → 2E + 5B + BE → PEC = C7

NOTA: Para calcular este checksum sólo se necesitan 3 bytes.

10.2.1. ECAN

Respecto al bus ECAN, las librerías se obtuvieron de Microchip, por lo que únicamente se han utilizado sus librerías adaptándolas al diseño.
Para poder utilizar las librerías de forma adecuada, se debe revisar el manual específico sobre ECAN detalladamente.

Por tanto, se recomienda para configurarlo:

$$BRP = \frac{FCAN}{[2*(N)TQ*Bit\ Rate]-1}$$

BRP = BaudRate Prescaler

FCAN = Frecuencia de reloj (CAN)

(N)TQ = Cantidad de tiempo en un bit

Bit Rate = Tasa de bits

```
/* CAN Baud Rate Configuration    */
#define FCAN     40000000
#define BITRATE  1000000
#define NTQ 20 //20 time quanta in a bit time
#define BRP_VAL ((FCAN/(2*NTQ*BitRate))-1)
```

NOTA: FCAN no puede superar los 40MHz

Dentro del este manual se recomienda seguir el siguiente procedimiento de configuración mínima:

- Paso 1: Modo de configuración de la solicitud desde el módulo ECAN.
- Paso 2: Seleccionar el reloj ECAN y el Bit Timming.
- Paso 3: Asignar el número de buffer dedicados al módulo ECAN en el espacio de memoria DMA.
- Paso 4: Configuración de los filtros y máscaras
- Paso 5: Poner el módulo ECAN en modo normal
- Paso 6: Configuración de los buffers de transmisión / recepción.

10.3. CÓDIGO DESARROLLADO

El proyecto está compuesto por una serie de archivos los cuales se han desarrollado en lenguaje de alto nivel C. El aspecto final que debe recoger el proyecto tiene que asemejarse al mostrado en la figura 10.3.1:

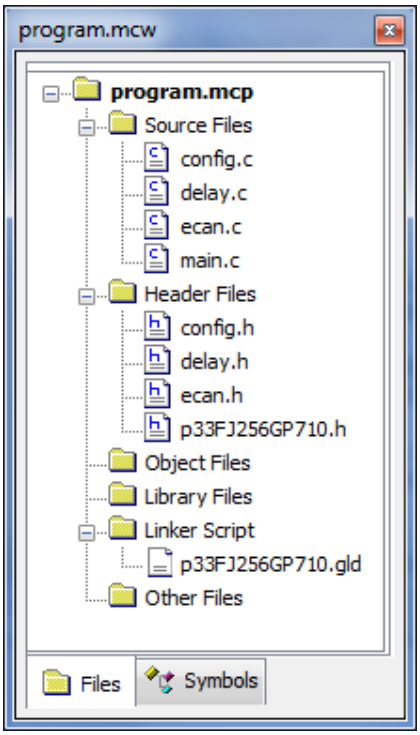


Figura 10.3.1

Los archivos se van a exponer tal y como se aprecian en la figura 10.3.1, es decir, en orden alfabético. Sin embargo, se van a omitir los que vienen por defecto integrados en MPLAB, como lo son p33FJ256GP710 de extensión '.h' y '.gld'.

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;       Configuration File
;       This file aims the microcontroler's configuration
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       Filename:               config.c
;       Date:                   Julio - 2012
;       File Version:           1.0
;       Assembled using:        MPLAB IDE 8.56.00.0
;       Author:                 Eduardo Mayoral
;       Company:                EUITT
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#include "config.h"

/*********************************************************************
* Function:         InitI2C()
* Input:            None.
* Output:           None.
* Overview:         Initialises the I2C(2) peripheral
* Note:             Sets up Master mode, No slew rate control, 50Khz
*********************************************************************/
unsigned int InitI2C(void)
{
        //This function will initialize the I2C(2) peripheral.
        //First set the I2C(2) BRG Baud Rate.

        //Consult the dSPIC Data Sheet for information on how to calculate the
        //baud Rate:        I2CBRG = [(1/Fscl - PGD) * Fcy] - 2

        I2C2BRG = 0x018f;           //0x0318 = 50KHz and 0x0188 = 100KHz

        //Now we will initialise the I2C peripheral for Master Mode, No Slew Rate
        //Control, and leave the peripheral switched off.

        I2C2CON = 0x1600;           //0b0001011000000000

        I2C2RCV = 0x0000;
        I2C2TRN = 0x0000;
        //Now we can enable the peripheral

        I2C2CON = 0x9200;
}

/*********************************************************************
* Function:         StartI2C()
* Input:            None.
* Output:           None.
* Overview:         Generates an I2C Start Condition
* Note:             None
*********************************************************************/
unsigned int StartI2C(void)
{
        //This function generates an I2C start condition and returns status
        //of the Start.

        I2C2CONbits.SEN = 1;                //Generate Start Condition
        while (I2C2CONbits.SEN);            //Wait for Start Condition
        //return(I2C1STATbits.S);           //Optionally return status
}
```

```
/*********************************************************************
* Function:         RestartI2C()
* Input:            None.
* Output:           None.
* Overview:         Generates a restart condition and optionally returns status
* Note:             None
*********************************************************************/
unsigned int RestartI2C(void)
{
        //This function generates an I2C Restart condition and returns status
        //of the Restart.

        I2C2CONbits.RSEN = 1;           //Generate Restart
        while (I2C2CONbits.RSEN);       //Wait for restart
        //return(I2C1STATbits.S);       //Optional - return status
}

/*********************************************************************
* Function:         StopI2C()
* Input:            None.
* Output:           None.
* Overview:         Generates a bus stop condition
* Note:             None
*********************************************************************/
unsigned int StopI2C(void)
{
        //This function generates an I2C stop condition and returns status
        //of the Stop.

        I2C2CONbits.PEN = 1;            //Generate Stop Condition
        while (I2C2CONbits.PEN);        //Wait for Stop
        //return(I2C1STATbits.P);       //Optional - return status
}

/*********************************************************************
* Function:         WriteI2C()
* Input:            Byte to write.
* Output:           None.
* Overview:         Writes a byte out to the bus
* Note:             None
*********************************************************************/
unsigned int WriteI2C(unsigned char byte)
{
        //This function transmits the byte passed to the function
        //while (I2C1STATbits.TRSTAT);  //Wait for bus to be idle
        I2C2TRN = byte;                 //Load byte to I2C1 Transmit buffer
        while (I2C2STATbits.TBF);       //wait for data transmission

}

/*********************************************************************
* Function:         IdleI2C()
* Input:            None.
* Output:           None.
* Overview:         Waits for bus to become Idle
* Note:             None
*********************************************************************/
unsigned int IdleI2C(void)
{
        while (I2C2STATbits.TRSTAT);            //Wait for bus Idle
}
```

```
/*****************************************************************
* Function:         ACKStatus()
* Input:            None.
* Output:           Acknowledge Status.
* Overview:         Return the Acknowledge status on the bus
* Note:             None
*****************************************************************/
unsigned int ACKStatus(void)
{
        return (!I2C2STATbits.ACKSTAT);         //Return Ack Status
}

/*****************************************************************
* Function:         NotAckI2C()
* Input:            None.
* Output:           None.
* Overview:         Generates a NO Acknowledge on the Bus
* Note:             None
*****************************************************************/
unsigned int NotAckI2C(void)
{
        I2C2CONbits.ACKDT = 1;                  //Set for NotACk
        I2C2CONbits.ACKEN = 1;
        while(I2C2CONbits.ACKEN);               //wait for ACK to complete
        I2C2CONbits.ACKDT = 0;                  //Set for NotACk
}

/*****************************************************************
* Function:         AckI2C()
* Input:            None.
* Output:           None.
* Overview:         Generates an Acknowledge.
* Note:             None
*****************************************************************/
unsigned int AckI2C(void)
{
        I2C2CONbits.ACKDT = 0;                  //Set for ACk
        I2C2CONbits.ACKEN = 1;
        while(I2C2CONbits.ACKEN);               //wait for ACK to complete
}

/*****************************************************************
* Function:         getsI2C()
* Input:            array pointer, Length.
* Output:           None.
* Overview:         read Length number of Bytes into array
* Note:             None
*****************************************************************/
unsigned int getsI2C(unsigned char *rdptr, unsigned char Length)
{
        while (Length --)
        {
                *rdptr++ = getI2C();            //get a single byte

                if(I2C2STATbits.BCL)            //Test for Bus collision
                {
                        return(-1);
                }

                if(Length)
                {
                        AckI2C();               //Acknowledge until all read
                }
        }
        return(0);
}
```

```
/*********************************************************************
* Function:         getI2C()
* Input:            None.
* Output:           contents of I2C1 receive buffer.
* Overview:         Read a single byte from Bus
* Note:             None
*********************************************************************/
unsigned int getI2C(void)
{
        I2C2CONbits.RCEN = 1;                   //Enable Master receive
        Nop();
        while(!I2C2STATbits.RBF);               //Wait for receive bufer to be full
        return(I2C2RCV);                        //Return data in buffer
}

/*********************************************************************
* Function:         EEAckPolling()
* Input:            Control byte.
* Output:           error state.
* Overview:         polls the bus for an Acknowledge from device
* Note:             None
*********************************************************************/
unsigned int EEAckPolling(unsigned char control)
{
        IdleI2C();                              //wait for bus Idle
        StartI2C();                             //Generate Start condition

        if(I2C2STATbits.BCL)
        {
                return(-1);                     //Bus collision, return
        }

        else
        {
                if(WriteI2C(control))
                {
                        return(-3);             //error return
                }

                IdleI2C();                      //wait for bus idle
                if(I2C2STATbits.BCL)
                {
                        return(-1);             //error return
                }

                while(ACKStatus())
                {
                        RestartI2C();           //generate restart
                        if(I2C2STATbits.BCL)
                        {
                                return(-1);     //error return
                        }

                        if(WriteI2C(control))
                        {
                                return(-3);
                        }

                        IdleI2C();
                }
        }
        StopI2C();                              //send stop condition
        if(I2C2STATbits.BCL)
        {
                return(-1);
        }
```

```
        return(0);
}

/*********************************************************************
* Function:         putstringI2C()
* Input:            pointer to array.
* Output:           None.
* Overview:         writes a string of data upto PAGESIZE from array
* Note:             None
********************************************************************/
unsigned int putstringI2C(unsigned char *wrptr)
{
        unsigned char x;

        for(x = 0; x < 32; x++)                 //Transmit Data Until Pagesize=32
        {
                if(WriteI2C(*wrptr))            //Write 1 byte
                {
                        return(-3);             //Return with Write Collision
                }
                IdleI2C();                      //Wait for Idle bus
                if(I2C2STATbits.ACKSTAT)
                {
                        return(-2);             //Bus responded with Not ACK
                }
                wrptr++;
        }
        return(0);
}

/*****************************************************************
* Function:         config()
*
* Overview:         Configures the registers of the application
*****************************************************************/
int config(void){

        //ADC outputs config (PORT B as output)
        TRISB = 0;                              //all PORTB as output
        AD1PCFGH = 0xff;                        //all PORTB as digital
        AD1PCFGL = 0xff;
        PORTB = 0xff;


        //TIMER1
        T1CONbits.TON = 0;                      //Timer1 desactivated
        TMR1 = 0;                               //Timer1 register is cleared
        T1CON = 0x8030;
        /*So...
        TON = 1;                                //Timer 1 is activated
        TCS = 0;                                //Main MCU serves as the source (Internal Clock)
        TCKPS = 11;                             //Prescaler is set to 1:256
        TGATE = 0;
        TSYNC = 0;
        TSIDL = 0;                              //By default
        ...Summarazing: T1CON = 1000 0000 0011 0000*/
        PR1=3600;                               //The timer adds up to reach the PR1 value
        IPC0bits.T1IP = 0x01;                   //Set Timer1 Interrrupt Priority Level
        IFS0bits.T1IF = 0;                      //Clear Timer1 Interrupt Flag
        IEC0bits.T1IE = 1;                      //Enable Timer1 Interrupt

        return 0;
}
```

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;       Delay File
;       The software supplied herewith by Microchip Technology Incorporated
;       (the “Company”) for its dsPIC30F Microcontroller is intended
;       and supplied to you, the Company’s customer, for use solely and
;       exclusively on Microchip's dsPIC30F Microcontroller products.
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       Filename:               delay.c
;       Date:                   Junio - 2006
;       File Version:           1.0
;       Assembled using:        MPLAB IDE 8.56.00.0
;       Author:                 Richard Fischer
;       Company:                Microchip
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/
#include "delay.h"

unsigned int temp_count;

#if defined(__dsPIC33F__)

void Delay( unsigned int delay_count )
{
        temp_count = delay_count +1;
        asm volatile("outer: dec _temp_count");
        asm volatile("cp0 _temp_count");
        asm volatile("bra z, done");
        asm volatile("do #3200, inner" );
        asm volatile("nop");
        asm volatile("inner: nop");
        asm volatile("bra outer");
        asm volatile("done:");
}


void Delay_Us( unsigned int delayUs_count )
{
        temp_count = delayUs_count +1;
        asm volatile("outer1: dec _temp_count");
        asm volatile("cp0 _temp_count");
        asm volatile("bra z, done1");
        asm volatile("do #1500, inner1" );
        asm volatile("nop");
        asm volatile("inner1: nop");
        asm volatile("bra outer1");
        asm volatile("done1:");
}
#elif defined(__PIC24H__)
void Delay( unsigned int delay_count )
{
        temp_count = delay_count +1;
        asm volatile("outer: dec _temp_count");
        asm volatile("cp0 _temp_count");
        asm volatile("bra z, done");
        asm volatile("repeat #3200");
        asm volatile("nop");
        asm volatile("repeat #3200");
        asm volatile("nop");
        asm volatile("bra outer");
        asm volatile("done:");
}
```

```
void Delay_Us( unsigned int delayUs_count )
{
        temp_count = delayUs_count +1;
        asm volatile("outer1: dec _temp_count");
        asm volatile("cp0 _temp_count");
        asm volatile("bra z, done1");
        asm volatile("repeat #1500");
        asm volatile("nop");
        asm volatile("repeat #1500");
        asm volatile("nop");
        asm volatile("bra outer1");
        asm volatile("done1:");
}

#endif
```

```
/********************************************************************************************
* © 2007 Microchip Technology Inc.
*
* FileName:        main.c
* Dependencies:    Header (.h) files if applicable, see below
* Processor:       dsPIC33Fxxxx
* Compiler:        MPLAB® C30 v3.00 or higher
*
* SOFTWARE LICENSE AGREEMENT:
* Microchip Technology Incorporated ("Microchip") retains all ownership and
* intellectual property rights in the code accompanying this message and in all
* derivatives hereto.  You may use this code, and any derivatives created by
* any person or entity by or on your behalf, exclusively with Microchip   ,s
* proprietary products.  Your acceptance and/or use of this code constitutes
* agreement to the terms and conditions of this notice.
*
* CODE ACCOMPANYING THIS MESSAGE IS SUPPLIED BY MICROCHIP "AS IS".  NO
* WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED
* TO, IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE APPLY TO THIS CODE, ITS INTERACTION WITH MICROCHIP   ,S
* PRODUCTS, COMBINATION WITH ANY OTHER PRODUCTS, OR USE IN ANY APPLICATION.
*
* YOU ACKNOWLEDGE AND AGREE THAT, IN NO EVENT, SHALL MICROCHIP BE LIABLE, WHETHER
* IN CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE OR BREACH OF STATUTORY DUTY),
* STRICT LIABILITY, INDEMNITY, CONTRIBUTION, OR OTHERWISE, FOR ANY INDIRECT,
* SPECIAL, PUNITIVE, EXEMPLARY, INCIDENTAL OR CONSEQUENTIAL LOSS, DAMAGE, FOR COST
* OR EXPENSE OF ANY KIND WHATSOEVER RELATED TO THE CODE, HOWSOEVER CAUSED, EVEN
* IF MICROCHIP HAS BEEN ADVISED OF THE POSSIBILITY OR THE DAMAGES ARE FORESEEABLE.
* TO THE FULLEST EXTENT ALLOWABLE BY LAW, MICROCHIP'S TOTAL LIABILITY ON ALL CLAIMS
* IN ANY WAY RELATED TO THIS CODE, SHALL NOT EXCEED THE PRICE YOU PAID DIRECTLY TO
* MICROCHIP SPECIFICALLY TO HAVE THIS CODE DEVELOPED.
*
* You agree that you are solely responsible for testing the code and
* determining its suitability.  Microchip has no obligation to modify, test,
* certify, or support the code.
*
* REVISION HISTORY:
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Author                Date      Comments on this revision
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
* Jatinder Gharoo   10/30/08  First release of source file
*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
********************************************************************************************/

/* Include the appropriate header (.h) file, depending on device used */

#if defined(__dsPIC33F__)
#include <p33Fxxxx.h>
#elif defined(__PIC24H__)
#include <p24hxxxx.h>
#endif

#include "ecan.h"

/************* START OF GLOBAL DEFINITIONS **********/


/************** END OF GLOBAL DEFINITIONS ***********/
```

```
/*****************************************************************************************
*   Function:       sendECAN
*   Description:    sends the message on a CAN bus
*   Arguments:      *message: a pointer to the message structure containing all the information about the message
*****************************************************************************************/
void sendECAN(mID *message)
{
        unsigned long word0=0;
        unsigned long word1=0;
        unsigned long word2=0;

        /*
        Message Format:
        Word0 : 0bUUUx xxxx xxxx xxxx
                                |_____________|||
                                              SID10:0   SRR IDE(bit 0)
        Word1 : 0bUUUU xxxx xxxx xxxx
                                          |_____________|
                                                        EID17:6
        Word2 : 0bxxxx xxx0 UUU0 xxxx
                                |_____||     |__|
                                EID5:0 RTR         DLC

        Remote Transmission Request Bit for standard frames
        SRR->   "0"         Normal Message
                            "1"  Message will request remote transmission
        Substitute Remote Request Bit for extended frames
        SRR->   should always be set to "1" as per CAN specification

        Extended  Identifier Bit
        IDE->   "0"  Message will transmit standard identifier
                            "1"  Message will transmit extended identifier

        Remote Transmission Request Bit for extended frames
        RTR->   "0"  Message transmitted is a normal message
                            "1"  Message transmitted is a remote message
        Don't care for standard frames
        */

        /* check to see if the message has an extended ID */
        if (message->frame_type==CAN_FRAME_EXT)
        {
                /* get the extended message id EID28..18*/
                word0=(message->id & 0x1FFC0000) >> 16;
                /* set the SRR and IDE bit */
                word0=word0+0x0003;
                /* the the value of EID17..6 */
                word1=(message->id & 0x0003FFC0) >> 6;
                /* get the value of EID5..0 for word 2 */
                word2=(message->id & 0x0000003F) << 10;
        }
        else
        {
                /* get the SID */
                word0=((message->id & 0x000007FF) << 2);
        }
        /* check to see if the message is an RTR message */
        if(message->message_type==CAN_MSG_RTR)
        {
                if(message->frame_type==CAN_FRAME_EXT)
                        word2=word2 | 0x0200;
                else
                        word0=word0 | 0x0002;

                ecan1msgBuf[message->buffer][0]=word0;
                ecan1msgBuf[message->buffer][1]=word1;
```

```
                ecan1msgBuf[message->buffer][2]=word2;
        }
        else
        {
                word2=word2+(message->data_length & 0x0F);
                ecan1msgBuf[message->buffer][0]=word0;
                ecan1msgBuf[message->buffer][1]=word1;
                ecan1msgBuf[message->buffer][2]=word2;
                /* fill the data */
                ecan1msgBuf[message->buffer][3]=((message->data[1] << 8) + message->data[0]);
                ecan1msgBuf[message->buffer][4]=((message->data[3] << 8) + message->data[2]);
                ecan1msgBuf[message->buffer][5]=((message->data[5] << 8) + message->data[4]);
                ecan1msgBuf[message->buffer][6]=((message->data[7] << 8) + message->data[6]);
        }
        /* set the message for transmission */
        C1TR01CONbits.TXREQ0=1;
}

/************************************************************************************
*   Function:       rxECAN
*   Description:    moves the message from the DMA memory to RAM
*   Arguments:      *message: a pointer to the message structure in RAM that will store the message.
************************************************************************************/
void rxECAN(mID *message)
{
        unsigned int ide=0;
        unsigned int rtr=0;
        unsigned long id=0;

        /*
        Standard Message Format:
        Word0 : 0bUUUx xxxx xxxx xxxx
                                |_____________|||
                                              SID10:0   SRR IDE(bit 0)
        Word1 : 0bUUUU xxxx xxxx xxxx
                                          |_____________|
                                                        EID17:6
        Word2 : 0bxxxx xxx0 UUU0 xxxx
                                |_____||      |__|
                                EID5:0 RTR          DLC
        word3-word6: data bytes
        word7: filter hit code bits

        Remote Transmission Request Bit for standard frames
        SRR->   "0"         Normal Message
                            "1"  Message will request remote transmission
        Substitute Remote Request Bit for extended frames
        SRR->   should always be set to "1" as per CAN specification

        Extended  Identifier Bit
        IDE->   "0"  Message will transmit standard identifier
                            "1"  Message will transmit extended identifier

        Remote Transmission Request Bit for extended frames
        RTR->   "0"  Message transmitted is a normal message
                            "1"  Message transmitted is a remote message
        Don't care for standard frames
        */

        /* read word 0 to see the message type */
        ide=ecan1msgBuf[message->buffer][0] & 0x0001;
```

```
		/* check to see what type of message it is */
		/* message is standard identifier */
		if(ide==0)
		{
			message->id=(ecan1msgBuf[message->buffer][0] & 0x1FFC) >> 2;
			message->frame_type=CAN_FRAME_STD;
			rtr=ecan1msgBuf[message->buffer][0] & 0x0002;
		}
		/* mesage is extended identifier */
		else
		{
			id=ecan1msgBuf[message->buffer][0] & 0x1FFC;
			message->id=id << 16;
			id=ecan1msgBuf[message->buffer][1] & 0x0FFF;
			message->id=message->id+(id << 6);
			id=(ecan1msgBuf[message->buffer][2] & 0xFC00) >> 10;
			message->id=message->id+id;
			message->frame_type=CAN_FRAME_EXT;
			rtr=ecan1msgBuf[message->buffer][2] & 0x0200;
		}
		/* check to see what type of message it is */
		/* RTR message */
		if(rtr==1)
		{
			message->message_type=CAN_MSG_RTR;
		}
		/* normal message */
		else
		{
			message->message_type=CAN_MSG_DATA;
			message->data[0]=(unsigned char)ecan1msgBuf[message->buffer][3];
			message->data[1]=(unsigned char)((ecan1msgBuf[message->buffer][3] & 0xFF00) >> 8);
			message->data[2]=(unsigned char)ecan1msgBuf[message->buffer][4];
			message->data[3]=(unsigned char)((ecan1msgBuf[message->buffer][4] & 0xFF00) >> 8);
			message->data[4]=(unsigned char)ecan1msgBuf[message->buffer][5];
			message->data[5]=(unsigned char)((ecan1msgBuf[message->buffer][5] & 0xFF00) >> 8);
			message->data[6]=(unsigned char)ecan1msgBuf[message->buffer][6];
			message->data[7]=(unsigned char)((ecan1msgBuf[message->buffer][6] & 0xFF00) >> 8);
			message->data_length=(unsigned char)(ecan1msgBuf[message->buffer][2] & 0x000F);
		}
		clearRxFlags(message->buffer);
}

/************************************************************************************
*	Function:		clearRxFlags
*	Description:	clears the rxfull flag after the message is read
*	Arguments:		buffer number to clear
************************************************************************************/
void clearRxFlags(unsigned char buffer_number)
{
		if((C1RXFUL1bits.RXFUL1) && (buffer_number==1))
			/* clear flag */
			C1RXFUL1bits.RXFUL1=0;
		/* check to see if buffer 2 is full */
		else if((C1RXFUL1bits.RXFUL2) && (buffer_number==2))
			/* clear flag */
			C1RXFUL1bits.RXFUL2=0;
		/* check to see if buffer 3 is full */
		else if((C1RXFUL1bits.RXFUL3) && (buffer_number==3))
			/* clear flag */
			C1RXFUL1bits.RXFUL3=0;
		else;

}
```

```
/*********************************************************************************
*   Function:       initCAN
*   Description:    Initialises the ECAN module
*   Arguments:      none
*********************************************************************************/
void initECAN (void)
{
        unsigned long temp;
        unsigned int tempint;

        /* put the module in configuration mode */
        C1CTRL1bits.REQOP=4;
        while(C1CTRL1bits.OPMODE != 4);

        /* FCAN is selected to be FCY
        FCAN = FCY = 40MHz */
        C1CTRL1bits.CANCKS = 0x1;

        /*
        Bit Time = (Sync Segment + Propagation Delay + Phase Segment 1 + Phase Segment 2)=20*TQ
        Phase Segment 1 = 8TQ
        Phase Segment 2 = 6Tq
        Propagation Delay = 5Tq
        Sync Segment = 1TQ
        CiCFG1<BRP> =(FCAN /(2 ×N×FBAUD))– 1
        BIT RATE OF 1Mbps
        */
        C1CFG1bits.BRP = BRP_VAL;
        /* Synchronization Jump Width set to 4 TQ */
        C1CFG1bits.SJW = 0x3;
        /* Phase Segment 1 time is 8 TQ */
        C1CFG2bits.SEG1PH=0x7;
        /* Phase Segment 2 time is set to be programmable */
        C1CFG2bits.SEG2PHTS = 0x1;
        /* Phase Segment 2 time is 6 TQ */
        C1CFG2bits.SEG2PH = 0x5;
        /* Propagation Segment time is 5 TQ */
        C1CFG2bits.PRSEG = 0x4;
        /* Bus line is sampled three times at the sample point */
        C1CFG2bits.SAM = 0x1;

        /* 4 CAN Messages to be buffered in DMA RAM */
        C1FCTRLbits.DMABS=0b000;

        /* Filter configuration */
        /* enable window to access the filter configuration registers */
        C1CTRL1bits.WIN=0b1;
        /* select acceptance mask 0 filter 0 buffer 1 */
        C1FMSKSEL1bits.F0MSK=0;
        /* configure accpetence mask 0 - match the id in filter 0
        setup the mask to check every bit of the standard message,
        the macro when called as CAN_FILTERMASK2REG_SID(0x7FF) will
        write the register C1RXM0SID to include every bit in filter comparison
        */
        C1RXM0SID=CAN_FILTERMASK2REG_SID(0x7FF);
        /* configure accpetence filter 0
        setup the filter to accept a standard id of 0x123,
        the macro when called as CAN_FILTERMASK2REG_SID(0x123) will
        write the register C1RXF0SID to accept only standard id of 0x123
        */
        C1RXF0SID=CAN_FILTERMASK2REG_SID(0x123);
        /* set filter to check for standard ID and accept standard id only */
        C1RXM0SID=CAN_SETMIDE(C1RXM0SID);
        C1RXF0SID=CAN_FILTERSTD(C1RXF0SID);
        /* acceptance filter to use buffer 1 for incoming messages */
        C1BUFPNT1bits.F0BP=0b0001;
```

```
/* enable filter 0 */
C1FEN1bits.FLTEN0=1;

/* select acceptance mask 1 filter 1 and buffer 2 */
C1FMSKSEL1bits.F1MSK=0b01;
/* configure accpetence mask 1 - match id in filter 1
setup the mask to check every bit of the extended message,
the macro when called as CAN_FILTERMASK2REG_EID0(0xFFFF)
will write the register C1RXM1EID to include extended
message id bits EID0 to EID15 in filter comparison.
the macro when called as CAN_FILTERMASK2REG_EID1(0x1FFF)
will write the register C1RXM1SID to include extended
message id bits EID16 to EID28 in filter comparison.
*/
C1RXM1EID=CAN_FILTERMASK2REG_EID0(0xFFFF);
C1RXM1SID=CAN_FILTERMASK2REG_EID1(0x1FFF);
/* configure acceptance filter 1
configure accpetence filter 1 - accept only XTD ID 0x12345678
setup the filter to accept only extended message 0x12345678,
the macro when called as CAN_FILTERMASK2REG_EID0(0x5678)
will write the register C1RXF1EID to include extended
message id bits EID0 to EID15 when doing filter comparison.
the macro when called as CAN_FILTERMASK2REG_EID1(0x1234)
will write the register C1RXF1SID to include extended
message id bits EID16 to EID28 when doing filter comparison.
*/
C1RXF1EID=CAN_FILTERMASK2REG_EID0(0x5678);
C1RXF1SID=CAN_FILTERMASK2REG_EID1(0x1234);
/* filter to check for extended ID only */
C1RXM1SID=CAN_SETMIDE(C1RXM1SID);
C1RXF1SID=CAN_FILTERXTD(C1RXF1SID);
/* acceptance filter to use buffer 2 for incoming messages */
C1BUFPNT1bits.F1BP=0b0010;
/* enable filter 1 */
C1FEN1bits.FLTEN1=1;

/* select acceptance mask 1 filter 2 and buffer 3 */
C1FMSKSEL1bits.F2MSK=0b01;
/* configure acceptance filter 2
configure accpetence filter 2 - accept only XTD ID 0x12345679
setup the filter to accept only extended message 0x12345679,
the macro when called as CAN_FILTERMASK2REG_EID0(0x5679)
will write the register C1RXF1EID to include extended
message id bits EID0 to EID15 when doing filter comparison.
the macro when called as CAN_FILTERMASK2REG_EID1(0x1234)
will write the register C1RXF1SID to include extended
message id bits EID16 to EID28 when doing filter comparison.
*/
C1RXF2EID=CAN_FILTERMASK2REG_EID0(0x5679);
C1RXF2SID=CAN_FILTERMASK2REG_EID1(0x1234);
/* filter to check for extended ID only */
C1RXF2SID=CAN_FILTERXTD(C1RXF2SID);
/* acceptance filter to use buffer 3 for incoming messages */
C1BUFPNT1bits.F2BP=0b0011;
/* enable filter 2 */
C1FEN1bits.FLTEN2=1;

/* clear window bit to access ECAN control registers */
C1CTRL1bits.WIN=0;

/* put the module in normal mode */
C1CTRL1bits.REQOP=0;
while(C1CTRL1bits.OPMODE != 0);

/* clear the buffer and overflow flags */
C1RXFUL1=C1RXFUL2=C1RXOVF1=C1RXOVF2=0x0000;
```

```
        /* ECAN1, Buffer 0 is a Transmit Buffer */
        C1TR01CONbits.TXEN0=1;
        /* ECAN1, Buffer 1 is a Receive Buffer */
        C1TR01CONbits.TXEN1=0;
        /* ECAN1, Buffer 2 is a Receive Buffer */
        C1TR23CONbits.TXEN2=0;
        /* ECAN1, Buffer 3 is a Receive Buffer */
        C1TR23CONbits.TXEN3=0;
        /* Message Buffer 0 Priority Level */
        C1TR01CONbits.TX0PRI=0b11;

        /* configure the device to interrupt on the receive buffer full flag */
        /* clear the buffer full flags */
        C1RXFUL1=0;
        C1INTFbits.RBIF=0;
}

/*****************************************************************************
*   Function:       initDMAECAN
*   Description:    Initialises the DMA to be used with ECAN module
                    Channel 0 of the DMA is configured to Tx ECAN messages of ECAN module 1.
                    Channel 2 is uconfigured to Rx ECAN messages of module 1.
*   Arguments:
*****************************************************************************/
void initDMAECAN(void)
{
        /* initialise the DMA channel 0 for ECAN Tx */
        /* clear the collission flags */
        DMACS0=0;
         /* setup channel 0 for peripheral indirect addressing mode
         normal operation, word operation and select as Tx to peripheral */
         DMA0CON=0x2020;
         /* setup the address of the peripheral ECAN1 (C1TXD) */
        DMA0PAD=0x0442;
        /* Set the data block transfer size of 8 */
        DMA0CNT=7;
        /* automatic DMA Tx initiation by DMA request */
        DMA0REQ=0x0046;
        /* DPSRAM atart adddress offset value */
        DMA0STA=__builtin_dmaoffset(&ecan1msgBuf);
        /* enable the channel */
        DMA0CONbits.CHEN=1;

        /* initialise the DMA channel 2 for ECAN Rx */
        /* clear the collission flags */
        DMACS0=0;
        /* setup channel 2 for peripheral indirect addressing mode
        normal operation, word operation and select as Rx to peripheral */
        DMA2CON=0x0020;
        /* setup the address of the peripheral ECAN1 (C1RXD) */
        DMA2PAD=0x0440;
        /* Set the data block transfer size of 8 */
        DMA2CNT=7;
        /* automatic DMA Rx initiation by DMA request */
        DMA2REQ=0x0022;
        /* DPSRAM atart adddress offset value */
        DMA2STA=__builtin_dmaoffset(&ecan1msgBuf);
        /* enable the channel */
        DMA2CONbits.CHEN=1;
}
```

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;         Main File
;         This file aims to develop the microcontroler's funtionality
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;         Filename:               main.c
;         Date:                   Julio - 2012
;         File Version:           1.0
;         Assembled using:        MPLAB IDE 8.56.00.0
;         Author:                 Eduardo Mayoral Ruiz
;         Company:                EUITT
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#include <p33FJ256GP710.h>
#include <stdio.h>
#include <string.h>
#include "config.h"
#include "ecan.h"
#include "delay.h"


/****************************************************************************************
Initial register's configuration:
_FOSCSEL: Start up with the FRC oscillator
_FOSC: Use Primary oscillator XT w/ PLL, Two-speed Oscillator Startup enabled
_FWDT: Watch dog timer is off
_FPOR: Power-on Reset Value disabled, Alternate I2C pins mapped to SDA1/SCL1
_FICD: JTAG is disabled,  communicate on PGC3/EMUC3 and PGD3/EMUD3 for explorer 16 development board
****************************************************************************************/
_FOSCSEL(FNOSC_FRC)
_FOSC(FCKSM_CSECMD & POSCMD_XT)
_FWDT(FWDTEN_OFF)
_FPOR(FPWRT_PWR1)
_FICD(JTAGEN_OFF & ICS_PGD1)


/****************************************************************************************
Defines
****************************************************************************************/
// CRC-8
#define GP  0x107                   // x^8 + x^2 + x + 1 --> CRC-8-CCITT
#define DI  0x07                    //0x07 normal; 0xE0 reversed; 0x83 reciprocal

// Sensor Commands
#define SENSOR1 0xb4                //Address of the Sensor1: 5A --> B4
#define SENSOR2 0xb6                //Address of the Sensor1: 5B --> B6
#define SENSOR3 0xb8                //Address of the Sensor1: 5C --> B8
#define SENSOR4 0xba                //Address of the Sensor1: 5D --> BA
#define RAM_ACCESS 0x00             //Use with a OR
#define EEPROM_ACCESS 0x20          //Use with a OR
#define READ_FLAGS 0xf0             //Use with a OR
#define SLEEP_MODE 0xff             //Use with a OR
#define RAM_TROOM 0x06              //Room Temperature located in the RAM
#define RAM_TOBJ1 0x07              //Object1 temperature located in the RAM
#define RAM_TOBJ2 0x08              //Object2 temperature located in the RAM

// Define ECAN Message Buffers
ECAN1MSGBUF ecan1msgBuf __attribute__((space(dma),aligned(ECAN1_MSG_BUF_LENGTH*16)));
```

```
/*********************************************************************************************
Globals
*********************************************************************************************/
mID canTxMessage;
mID canRxMessage;
unsigned char varT1=1;
unsigned char ControlByte=0x07;                 //Control Byte or Command
unsigned char Data=0x00;                        //Data Byte
unsigned char Length=0x01;                      //Length of Bytes to Read
unsigned char dat_LOW = 0x05;
unsigned char dat_HIGH = 0x06;
unsigned char PEC = 0xff;
unsigned char SlaveAddress = 0xba;              //Address sensor by default 5B shift-left
unsigned int Temp_aux=0;
float tempEnv_S1=0, tempEnv_S2=0, tempEnv_S3=0, tempEnv_S4=0;
float tempO1_S1=0, tempO1_S2=0, tempO1_S3=0, tempO1_S4=0;
//float tempO2_S1=0, tempO2_S2=0, tempO2_S3=0, tempO2_S4=0;
unsigned char made_table=0;
unsigned char crc=0x00;
unsigned char crc8_table[256];                  //8-bit table

/*********************************************************************************************
Function declarations
*********************************************************************************************/
unsigned int read_sensor(unsigned char ControlByte, unsigned char SlaveAddress, unsigned char *Data, unsigned
char Length);
float calc_temperature(unsigned char dat_HIGH, unsigned char dat_LOW);
void init_crc8();
void crc8(unsigned char *crc, unsigned char m);
void delay();
void oscConfig();


/*********************************************************************************************
TIMER1 Interruption Routine
*********************************************************************************************/
void __attribute__ ((__interrupt__, no_auto_psv)) _T1Interrupt(void)
{
        T1CONbits.TON = 0;                      //Stop T1
        TMR1 = 0;                               //Clear T1 counter
        varT1 = 0;
        IFS0bits.T1IF = 0;                      //Clear Timer1 Interrupt Flag
}

/*********************************************************************************************
MAIN FUNCTION
*********************************************************************************************/
int main (void)
{
        unsigned char repeat=1;                 //To just take one measurement in each iteration
        unsigned char object=1, i=3, I2C_ECAN=1;
        unsigned char sensors[8];

        //Initialization
        oscConfig();
        config();
        initECAN();
        initDMAECAN();
        InitI2C();

        T1CONbits.TON=0;
        init_crc8();

        for(i=0;i<8;i++){
                sensors[i]=0;
        }
```

```
i=3;
canTxMessage.id=0x123;                  //Identifier can start with any number

//Enable ECAN1 Interrupt
IEC2bits.C1IE=1;
//Enable Transmit interrupy
C1INTEbits.TBIE=1;
//Enable Receive interrupt
C1INTEbits.RBIE=1;


while(1){

        if(I2C_ECAN==1){                //If I2C_ECAN == I2C
                LATAbits.LATA0 = 1;

                if (i==0)
                {
                        SlaveAddress = SENSOR1;
                        read_sensor(ControlByte, SlaveAddress, &Data, 1);

                        if (object==1)
                        {
                                tempO1_S1 = calc_temperature(dat_HIGH, dat_LOW);
                                sensors[0]=dat_LOW;
                                sensors[1]=dat_HIGH;
                        }
                        else if (object==2) tempEnv_S1 = calc_temperature(dat_HIGH, dat_LOW);

                        if (object==1 && repeat==1)
                        {
                                ControlByte=0x06; //Read enviroment temperature
                                object=2;
                        }
                        else if (object==2 && repeat==1)
                        {
                                ControlByte=0x07; //Read object temperature
                                object=1;
                                I2C_ECAN=0;
                        }
                        repeat = 0;
                        i=3;

                }
                if (i==1)
                {
                        SlaveAddress = SENSOR2;
                        read_sensor(ControlByte, SlaveAddress, &Data, 1);

                        if (object==1)
                        {
                                tempO1_S2 = calc_temperature(dat_HIGH, dat_LOW);
                                sensors[2]=dat_LOW;
                                sensors[3]=dat_HIGH;
                        }
                        else if (object==2) tempEnv_S2 = calc_temperature(dat_HIGH, dat_LOW);

                        i--;
                }
                if (i==2)
                {
                        SlaveAddress = SENSOR3;
                        read_sensor(ControlByte, SlaveAddress, &Data, 1);
```

```
                        if (object==1)
                        {
                                tempO1_S3 = calc_temperature(dat_HIGH, dat_LOW);
                                sensors[4]=dat_LOW;
                                sensors[5]=dat_HIGH;
                        }
                        else if (object==2) tempEnv_S3 = calc_temperature(dat_HIGH, dat_LOW);

                        i--;
                }
                if (i==3 && repeat==1)
                {
                        SlaveAddress = SENSOR4;
                        read_sensor(ControlByte, SlaveAddress, &Data, 1);

                        if (object==1)
                        {
                                tempO1_S4 = calc_temperature(dat_HIGH, dat_LOW);
                                sensors[6]=dat_LOW;
                                sensors[7]=dat_HIGH;
                        }
                        else if (object==2) tempEnv_S4 = calc_temperature(dat_HIGH, dat_LOW);

                        i--;
                }
                Nop();
                LATAbits.LATA0 = 0;
                repeat = 1;
                crc=0x00;
                delay();
}//end if(I2C)


if(I2C_ECAN==0)                         //If I2C_ECAN == ECAN
{
//Check when a message is received and move the message into RAM and parse the message

                if(canRxMessage.buffer_status==CAN_BUF_FULL)
                {
                        rxECAN(&canRxMessage);

                        //Reset the flag when done
                        canRxMessage.buffer_status=CAN_BUF_EMPTY;
                }
                else
                {
                        // delay for one second
                        // Delay(1);
                        // send another message

                        //Send the stored information from sensors
                        //Configure and send a message
                        canTxMessage.message_type=CAN_MSG_DATA;
                        //canTxMessage.message_type=CAN_MSG_RTR;
                        canTxMessage.frame_type=CAN_FRAME_EXT;
                        //canTxMessage.frame_type=CAN_FRAME_STD;

                        canTxMessage.buffer=0;
                        canTxMessage.data[0]=sensors[0];
                        canTxMessage.data[1]=sensors[1];
                        canTxMessage.data[2]=sensors[2];
                        canTxMessage.data[3]=sensors[3];
                        canTxMessage.data[4]=sensors[4];
                        canTxMessage.data[5]=sensors[5];
                        canTxMessage.data[6]=sensors[6];
                        canTxMessage.data[7]=sensors[7];
```

```
                                canTxMessage.data_length=8;

                                //Delay for a second
                                Delay(1);

                                canTxMessage.id++;

                                //Send a CAN message
                                sendECAN(&canTxMessage);
                        }

                        I2C_ECAN=1;

                }//end if(ECAN)

        }//end while(1)

return 0;
}//end MAIN


/*****************************************************************************************************
read_sensor() function
*****************************************************************************************************/
unsigned int read_sensor(unsigned char ControlByte, unsigned char SlaveAddress, unsigned char *Data, unsigned
char Length)
{
        IdleI2C();                      //Wait for bus Idle
        StartI2C();                     //Generate Start Condition
        WriteI2C(SlaveAddress);         //Write start address
        crc8(&crc, SlaveAddress);
        IdleI2C();                      //Wait for bus Idle
        WriteI2C(ControlByte);          //Write Control Byte
        crc8(&crc, ControlByte);
        IdleI2C();                      //Wait for bus Idle

        RestartI2C();                   //Generate restart condition
        WriteI2C(SlaveAddress | 0x01);  //Write control byte for read
        crc8(&crc, SlaveAddress | 0x01);
        IdleI2C();                      //Wait for bus Idle

        getsI2C(Data, Length);          //Read the first byte
        dat_LOW = *Data;
        crc8(&crc, dat_LOW);
        NotAckI2C();
        getsI2C(Data, Length);          //Read the second byte
        dat_HIGH = *Data;
        crc8(&crc, dat_HIGH);
        NotAckI2C();
        getsI2C(Data, Length);          //Read PEC (checksum)
        PEC = *Data;
        if (crc == PEC) AckI2C();       //Send NACK if the checksum is ok to finish the transmission
        else NotAckI2C();
        StopI2C();                      //Generate Stop condition

        return (0);
}
```

```
/*****************************************************************************************
calc_temperature() function
*****************************************************************************************/
float calc_temperature(unsigned char dat_HIGH, unsigned char dat_LOW)
{
	Temp_aux = dat_HIGH;
	Temp_aux = Temp_aux << 8;
	Temp_aux = Temp_aux + dat_LOW;

	return ((float)Temp_aux/50-273.15);
}



/*****************************************************************************************
CRC8 functions
*****************************************************************************************/
void init_crc8()
/* Should be called before any other crc function */
{
	int i,j;
	unsigned char crc;

	if (!made_table) {
		for (i=0; i<256; i++) {
			crc = i;
			for (j=0; j<8; j++)
				crc = (crc << 1) ^ ((crc & 0x80) ? DI : 0);
			crc8_table[i] = crc & 0xFF;
		}
	made_table=1;
	}
}

void crc8(unsigned char *crc, unsigned char m)
/*	For a byte array whose accumulated crc value is stored in *crc, computes
	resultant crc obtained by appending m to the byte array */
{
	if (!made_table)
	init_crc8();

	*crc = crc8_table[(*crc) ^ m];
	*crc &= 0xFF;
}



/*****************************************************************************************
delay() function
*****************************************************************************************/
void delay()
{	//Look over PR1 (in config.h) to see the number to be reached in the account
	T1CONbits.TON = 1;
	while(varT1);		//When it enters into the interrupt, it goes out from this loop
	varT1=1;
}
```

```
/*********************************************************************************************
oscConfig() function
*********************************************************************************************/
void oscConfig(void){

        /*  Configure Oscillator to operate the device at 40MIPS
        Fosc= Fin*M/(N1*N2), Fcy=Fosc/2
        Fosc= 8MHz*40/(2*2)=80Mhz for 8MHz input clock */

        PLLFBD=38;                              // M=40 => PLLFD=38
        CLKDIVbits.PLLPOST=0;                   // N1=2
        CLKDIVbits.PLLPRE=0;                    // N2=2
        OSCTUN=0;                               // Tune FRC oscillator, if FRC is used

        //Disable Watch Dog Timer
        RCONbits.SWDTEN=0;
        //Clock switch to incorporate PLL
        __builtin_write_OSCCONH(0x03);          // Initiate Clock Switch to Primary
                                                // Oscillator with PLL (NOSC=0b011)
        __builtin_write_OSCCONL(0x01);          // Start clock switching
        while (OSCCONbits.COSC != 0b011);       // Wait for Clock switch to occur

        //Wait for PLL to lock
        while(OSCCONbits.LOCK!=1) {};
}

/****** START OF INTERRUPT ECAN SERVICE ROUTINES ******/
void __attribute__((interrupt,no_auto_psv))_C1Interrupt(void)
{
        /* check to see if the interrupt is caused by receive */
   if(C1INTFbits.RBIF)
   {
             /* check to see if buffer 1 is full */
             if(C1RXFUL1bits.RXFUL1)
             {
                        /* set the buffer full flag and the buffer received flag */
                        canRxMessage.buffer_status=CAN_BUF_FULL;
                        canRxMessage.buffer=1;
                }
                /* check to see if buffer 2 is full */
                else if(C1RXFUL1bits.RXFUL2)
                {
                        /* set the buffer full flag and the buffer received flag */
                        canRxMessage.buffer_status=CAN_BUF_FULL;
                        canRxMessage.buffer=2;
                }
                /* check to see if buffer 3 is full */
                else if(C1RXFUL1bits.RXFUL3)
                {
                        /* set the buffer full flag and the buffer received flag */
                        canRxMessage.buffer_status=CAN_BUF_FULL;
                        canRxMessage.buffer=3;
                }
                else;
                /* clear flag */
                C1INTFbits.RBIF = 0;
        }
        else if(C1INTFbits.TBIF)
        {
                 /* clear flag */
                C1INTFbits.TBIF = 0;
        }
        else;       /* clear interrupt flag */
        IFS2bits.C1IF=0;
}
/********* END OF INTERRUPT ECAN SERVICE ROUTINES ********/
```

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;       Configuration File
;       This file aims the microcontroler's configuration
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       Filename:               config.h
;       Date:                   Julio - 2012
;       File Version:           1.0
;       Assembled using:        MPLAB IDE 8.56.00.0
;       Author:                 Eduardo Mayoral
;       Company:                EUITT
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#include <p33FJ256GP710.h>

//Low Level Functions
unsigned int IdleI2C(void);
unsigned int StartI2C(void);
unsigned int WriteI2C(unsigned char);
unsigned int StopI2C(void);
unsigned int RestartI2C(void);
unsigned int getsI2C(unsigned char*, unsigned char);
unsigned int NotAckI2C(void);
unsigned int InitI2C(void);
unsigned int ACKStatus(void);
unsigned int getI2C(void);
unsigned int AckI2C(void);
unsigned int EEAckPolling(unsigned char);
unsigned int putstringI2C(unsigned char*);
int config(void);
```

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;       Delay File
;       The software supplied herewith by Microchip Technology Incorporated
;       (the “Company”) for its dsPIC30F Microcontroller is intended
;       and supplied to you, the Company’s customer, for use solely and
;       exclusively on Microchip's dsPIC30F Microcontroller products.
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       Filename:               delay.h
;       Date:                   Junio - 2006
;       File Version:           1.0
;       Assembled using:        MPLAB IDE 8.56.00.0
;       Author:                 Richard Fischer
;       Company:                Microchip
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#define Fcy  40000000

void Delay( unsigned int delay_count );
void Delay_Us( unsigned int delayUs_count );

#define Delay200uS_count    (Fcy * 0.0002) / 1080
#define Delay_1mS_Cnt       (Fcy * 0.001) / 2950
#define Delay_2mS_Cnt       (Fcy * 0.002) / 2950
#define Delay_5mS_Cnt       (Fcy * 0.005) / 2950
#define Delay_15mS_Cnt      (Fcy * 0.015) / 2950
#define Delay_1S_Cnt        (Fcy * 1) / 2950
```

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;
;       Ecan File
;       The software supplied herewith by Microchip Technology Incorporated
;       (the “Company”) for its dsPIC30F Microcontroller is intended
;       and supplied to you, the Company’s customer, for use solely and
;       exclusively on Microchip's dsPIC30F Microcontroller products.
;
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
;       Filename:               ecan.h
;       Date:                   Enero - 2007
;       File Version:           1.0
;       Assembled using:        MPLAB IDE 8.56.00.0
;       Author:                 Jatinder Gharoo
;       Company:                Microchip
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~*/

#ifndef __ecan_H
#define __ecan_H

/* CAN Baud Rate Configuration*/
#define FCAN            40000000
#define BITRATE         1000000
#define NTQ             20              // 20 Time Quanta in a Bit Time
#define BRP_VAL         ((FCAN/(2*NTQ*BITRATE))-1)

/* defines used in ecan.c */
/* ECAN message buffer length */
#define ECAN1_MSG_BUF_LENGTH 4

/* ECAN message type identifiers */
#define CAN_MSG_DATA  0x01
#define CAN_MSG_RTR   0x02
#define CAN_FRAME_EXT 0x03
#define CAN_FRAME_STD 0x04
#define CAN_BUF_FULL  0x05
#define CAN_BUF_EMPTY 0x06

/* filter and mask defines */
/* Macro used to write filter/mask ID to Register CiRXMxSID and
CiRXFxSID. For example to setup the filter to accept a value of
0x123, the macro when called as CAN_FILTERMASK2REG_SID(0x123) will
write the register space to accept message with ID ox123
USE FOR STANDARD MESSAGES ONLY */
#define CAN_FILTERMASK2REG_SID(x) ((x & 0x07FF)<< 5)
/* the Macro will set the "MIDE" bit in CiRXMxSID */
#define CAN_SETMIDE(sid) (sid | 0x0008)
/* the macro will set the EXIDE bit in the CiRXFxSID to
accept extended messages only */
#define CAN_FILTERXTD(sid) (sid | 0x0008)
/* the macro will clear the EXIDE bit in the CiRXFxSID to
accept standard messages only */
#define CAN_FILTERSTD(sid) (sid & 0xFFF7)
/* Macro used to write filter/mask ID to Register CiRXMxSID, CiRXMxEID and
CiRXFxSID, CiRXFxEID. For example to setup the filter to accept a value of
0x123, the macro when called as CAN_FILTERMASK2REG_SID(0x123) will
write the register space to accept message with ID 0x123
USE FOR EXTENDED MESSAGES ONLY */
#define CAN_FILTERMASK2REG_EID0(x) (x & 0xFFFF)
#define CAN_FILTERMASK2REG_EID1(x) (((x & 0x1FFC)<< 3)|(x & 0x3))

/* DMA CAN Message Buffer Configuration */
typedef unsigned int ECAN1MSGBUF [ECAN1_MSG_BUF_LENGTH][8];
extern ECAN1MSGBUF  ecan1msgBuf __attribute__((space(dma)));
```

```
/* message structure in RAM */
typedef struct{
        /* keep track of the buffer status */
        unsigned char buffer_status;
        /* RTR message or data message */
        unsigned char message_type;
        /* frame type extended or standard */
        unsigned char frame_type;
        /* buffer being used to send and receive messages */
        unsigned char buffer;
        /* 29 bit id max of 0x1FFF FFFF
           11 bit id max of 0x7FF */
        unsigned long id;
        unsigned char data[8];
        unsigned char data_length;
}mID;

/* function prototypes as defined in can.c */
void initECAN (void);
void sendECAN(mID *message);
void initDMAECAN(void);
void rxECAN(mID *message);
void clearRxFlags(unsigned char buffer_number);

#endif
```

10.3.1. Modificación de la dirección SlaveAddress en el dispositivo MLX90614

También se añaden en esta memoria las funciones que se desarrollaron para modificar las direcciones SlaveAddress de los sensores. Téngase en cuenta que los sensores se han de modificar uno por uno, por lo que el código sólo es válido para una modificación, para los demás se deberán variar los datos clave a escribir en el dispositivo.
Además, se recuerda que para poder escribir en el sensor primero se tiene que borrar el contenido de la dirección de la EPROM 0x00E → Command = 0x2E.
Veáse como se cambia la dirección 5A por 5D:

```
unsigned int clear_SA(unsigned char ControlByte, unsigned char Address, unsigned char LowAdd, unsigned char HighAdd, unsigned char PEC)
{
	IdleI2C();			//Wait for bus Idle
	StartI2C();			//Generate Start Condition
	WriteI2C(0x00);			//Write start address
	IdleI2C();			//Wait for bus Idle
	WriteI2C(0x2e);			//Write Control Byte
	IdleI2C();			//Wait for bus Idle

	WriteI2C(0x00);
	IdleI2C();

	WriteI2C(0x00);
	IdleI2C();

	WriteI2C(0x6f);			//Write PEC
	IdleI2C();

	StopI2C();			//Generate Stop

	return (0);
}

unsigned int write_sensor(unsigned char ControlByte, unsigned char Address, unsigned char LowAdd, unsigned char HighAdd, unsigned char PEC)
{
	IdleI2C();
	StartI2C();
	WriteI2C(0x00);			//SlaveAddress
	IdleI2C();
	WriteI2C(0x2e);			//Write Control Byte
	IdleI2C();

	WriteI2C(0x5d);			//Write new SlaveAddress (LowByte)
	IdleI2C();

	WriteI2C(0x00);			//Write new SlaveAddress (HighByte)
	IdleI2C();

	WriteI2C(0x8a);			//Write PEC
	IdleI2C();

	StopI2C();			//Generate Stop

	return (0);
}
```

Estas funciones dentro de una función *main* se gestionarán de la siguiente manera:

```
int main (void)
{
        […]

        clear_SA(0x2e, 0x00, 0x00, 0x00, 0x6f);

        Delay(1s); //As minimun 200 miliseconds

        write_sensor(0x02e, 0x00, 0x5d, 0x00, 0x8a);

        Delay(1s); //As minimun 200 miliseconds

        […]
}
```

10.4. Diagrama de la función main{}

Para ofrecer una visión general y más clara de la funcionalidad de la aplicación, se presenta a continuación en la figura 10.4.1 el diagrama de la función main{}:

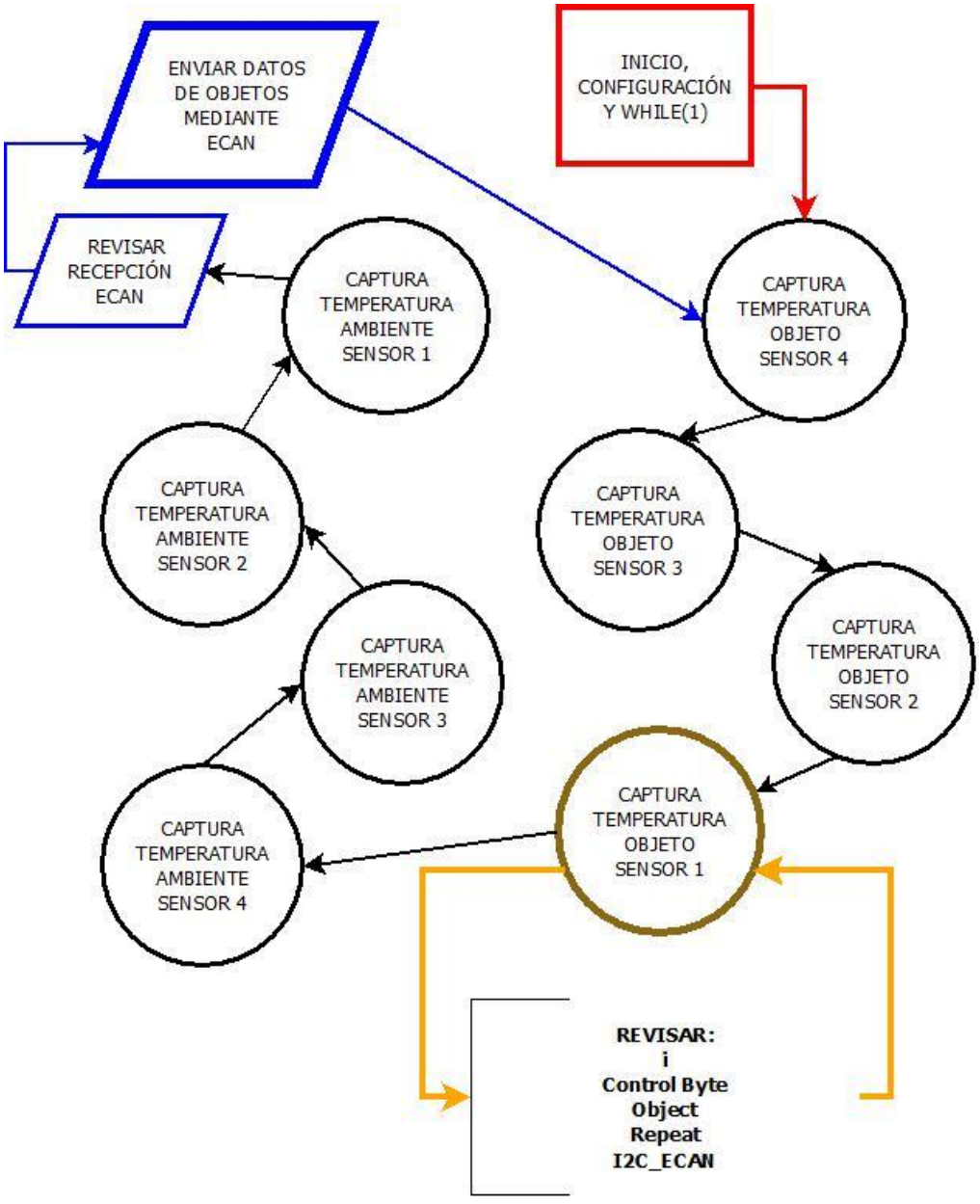


Figura 10.4.1

Simplemente, se adquieren las temperaturas por I2C y una vez recogidas todas, se envían mediante el bus CAN las registradas de los objetos en una única trama. Este proceso se repite mediante un bucle infinito de tipo *while*.

11. PRESUPUESTO

En la siguiente tabla se aprecia un detallado desglose de los componentes utilizados, especificándose su encapsulado y su precio:

11.1. PRECIO DE LOS COMPONENTES

Tipo de componente	Componente	Encapsulado	Cantidad	Precio/Unidad	Precio/conjunto
Circuitos Integrados					
Microcontrolador	dsPIC33FJ256MC710A-I/PF	TQFP SMD	1	11,02	11,02
Transceiver canBUS	MCP2551SN	SO-08 SMD	1	1,18	1,18
Memoria Flash	M25P80	SO-08 SMD	1	1,86	1,86
Conversor DC-DC	LM2576T-12V_5V	TO-220	1	3,87	3,87
Conversor DC-DC	LM1117T-5V_3,3V	SOT223	1	1,39	1,39
Conectores					
MemoriaSD	Socket MOLEX	6173PACK	1	3,51	3,51
Con.Interconexión	Conector clema a tornillo		6	0,5	3
Fila debug de 6 pines	Pines para Pickit2	Pin	1	0,6	0,6
Con. debug de 6 pines	RJ11-6X	RJ11	1	0,26	0,26
Con. busCAN	RS232	DB9	1	0,5	0,5
Con. Sensores I2C	Conector Cuadrado 4 pines		4	0,5	2
Resistores					
100 omhs	R100	1206 SMD	2	0,05	0,1
170 omhs	R170	1206 SMD	1	0,096	0,096
470 omhs	R470	1206 SMD	1	0,096	0,096
4700 omhs	R4k7	1206 SMD	1	0,11	0,11
10 komhs	R10k	1206 SMD	9	0,022	0,198
33 komhs	R33k	1206 SMD	1	0,11	0,11
100 komhs	R100k	1206 SMD	6	0,11	0,66
1 Momhs	R1M	1206 SMD	1	0,11	0,11
Condensadores					
22 pFaradios	C22p	E2-4	5	0,31	1,55
47 nFaradios	C47n	E2-4	5	0,055	0,275
0,1 uFaradios	C100n	E2-4	10	0,175	1,75
0,1 uFaradios	C100n	1206 SMD	1	0,022	0,022
10 uFaradios	C10u	E2-4	5	0,05	0,25
100 uFaradios	C100u	E2-4	5	0,064	0,32
1000 uFaradios	C1m	E2-4	1	0,69	0,69
Otros					
Diodo rectificador	1N5822	DO35Z10	2	0,32	0,64
Bobina 100 uHenrios	L100u	E2-4	1	0,37	0,37
Oscilador 32kHz	OSC 32kHz	QS	1	1,77	1,77
Oscilador 8 MHz	OSC 8MHz	QS	1	0,45	0,45
Diodo LED	Led azul	E2-4	1	0,45	0,45
Pulsador	Switch_pulse	B3F-10	1	0,68	0,68
Sensor IR	Termopila IR PWM	TO-39	4	10,04	40,16
TOTAL componentes (incluyendo el IVA)					**69,03**

11.2. PRECIO DE LA TARJETA

Para realizar la tarjeta de forma no profesional será necesario lo siguiente:

Elemento	Tipo	Encapsulado	Dimensiones	Precio
Placa de cobre	Fotosensible doble cara	Plancha	100x160 DC	5,21
Soldador de mano	Soldador punta fina		30W/230V	4,5
Flux	Flux liquido	Bote	150ml	3
Estaño	Estaño para soldar	Rollo	1m	4
TOTAL elementos (incluyendo el IVA)				**16,71**

BatchPCB

Sin embargo, siempre se puede recurrir a una empresa de fabricación automática para que ellos implementen el diseño. Simplemente se les debe proporcionar los archivos gerbers (los cuales anteriormente se explicaba como crearlos) y solicitar un presupuesto sin compromiso, realizándose cómodamente mediante la página web de dichos fabricantes.

En este diseño se recurre a BatchPCB para solicitar un presupuesto, donde en las siguientes figuras se muestran los resultados proporcionados por dicha empresa:

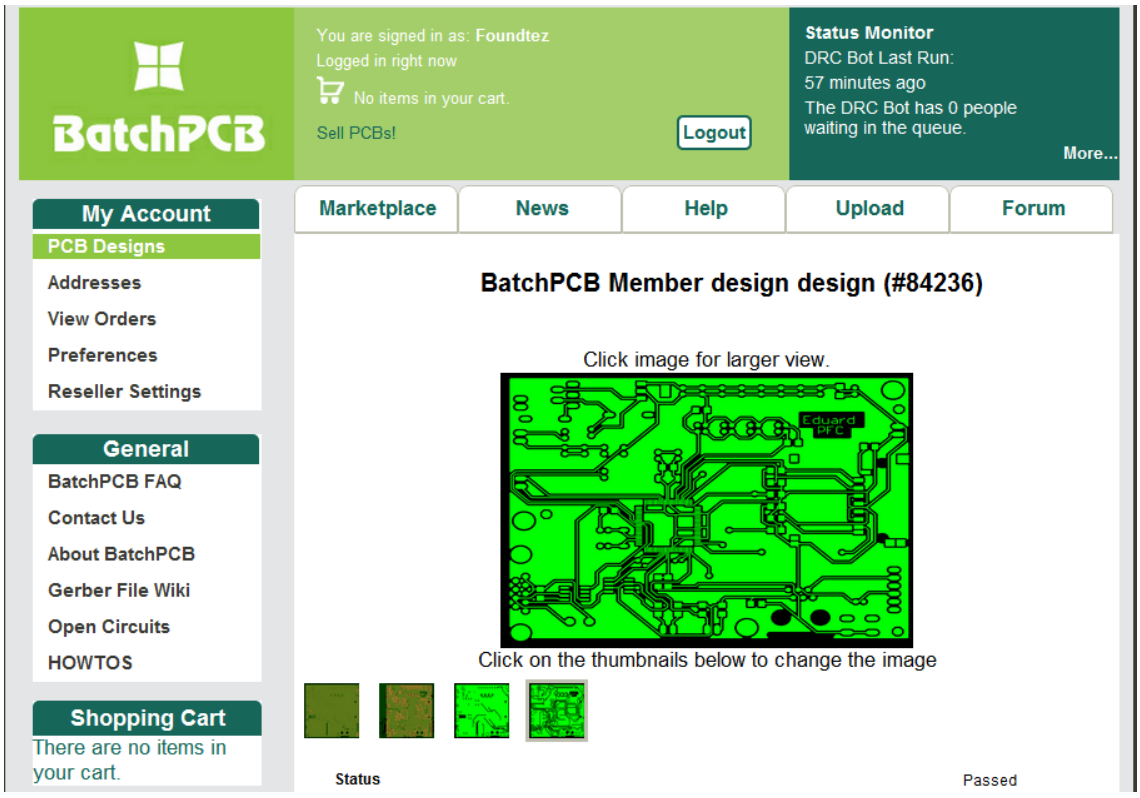


Figura 11.2.1

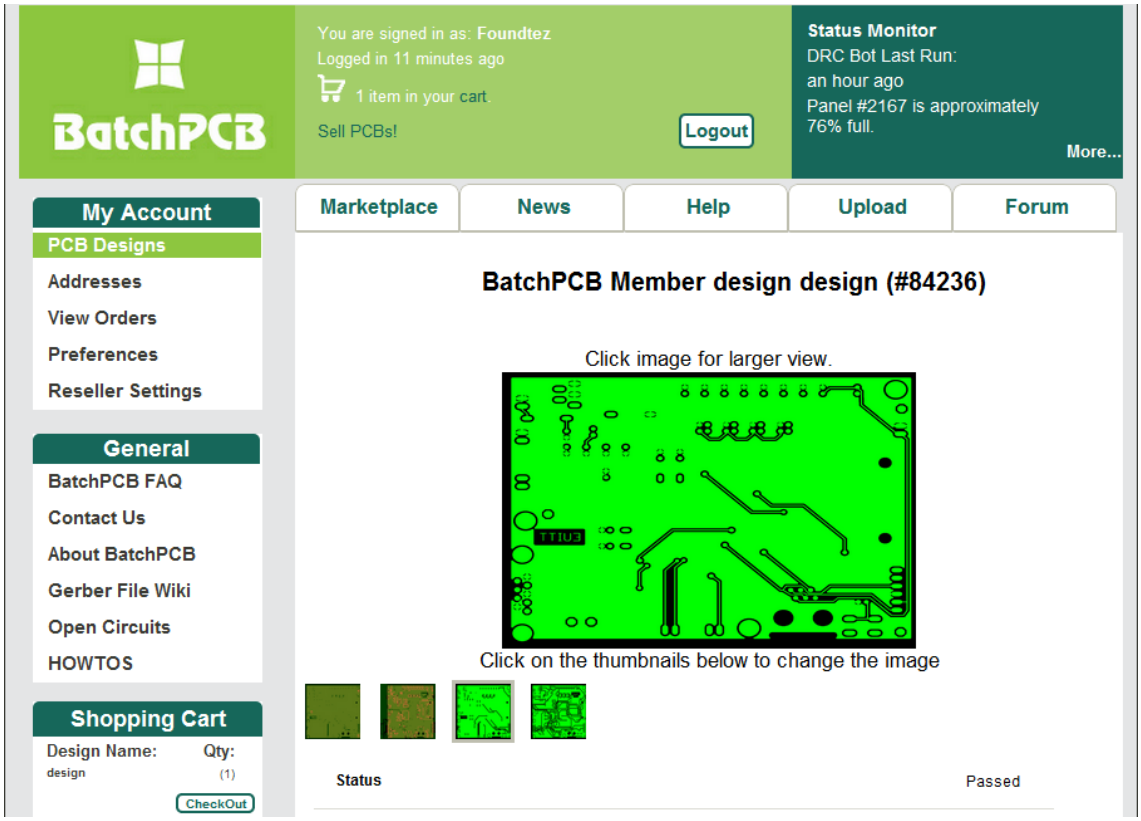


Figura 11.2.2

En la figura 11.1.3 se presenta el presupuesto completo ofrecido por BatchPBC:

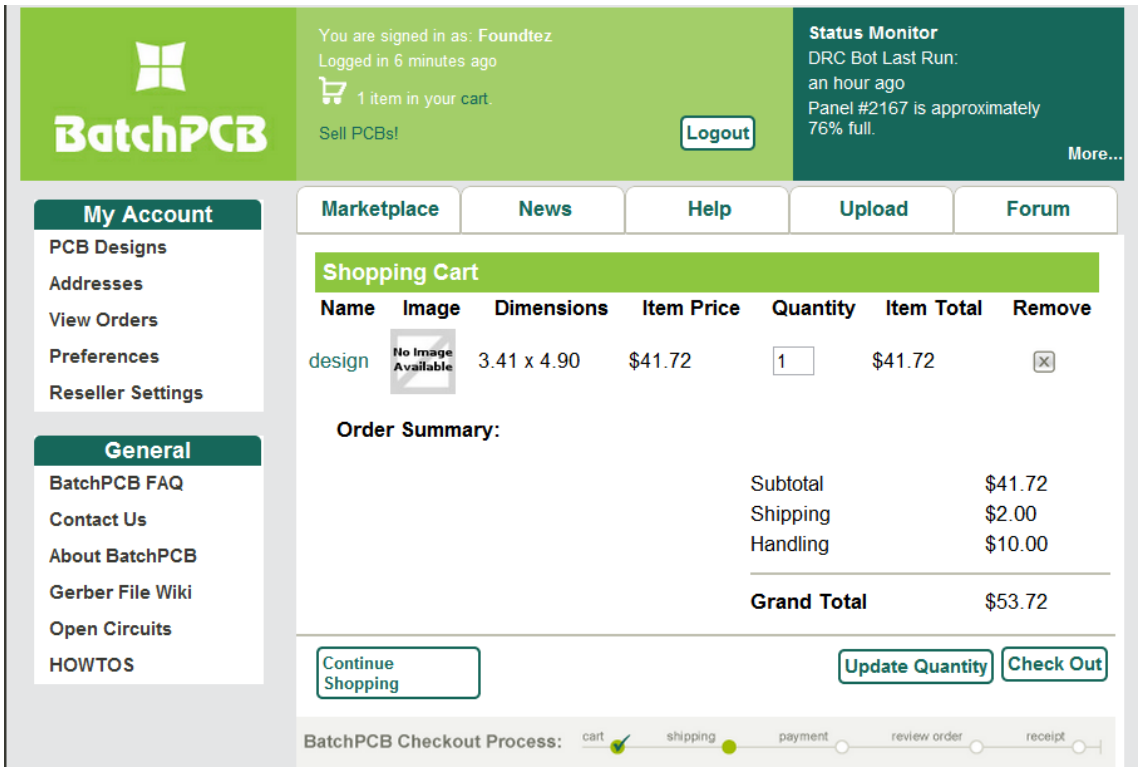


Figura 11.2.3

De este modo podemos distinguir los dos métodos de implementar la tarjeta:

Método no profesional

El presupuesto definitivo de este método, que ha sido con el que se ha procedido para la implementación de este proyecto, es el siguiente:

Componentes	69,03€
Elementos para realizar la PCB	16,71€
TOTAL	**85,74€**

NOTA: Téngase en cuenta que **no se han añadido los gastos de envío** de los componentes, ya que es un factor que varía dependiente tanto de la ubicación como de la empresa a la cual se le soliciten.

Método profesional

El presupuesto definitivo de este método, que ha sido descartado para la implementación de este proyecto, es el siguiente:

Componentes		69,03€
Presupuesto de BatchPCB	53,72$ =	42,91€
TOTAL		111,94€

Diferencia entre métodos: 111,94 – 85,74 = 26,2€

Y por último, se añade un cálculo aproximado de lo que costaría realizar este proyecto de forma profesional, es decir, lo que hubiera costado contratar a un diseñador si este proyecto no tuviese fines académicos:

Nº de horas dedicadas al diseño:	6h/día durante 4 meses = 480 horas
Salario del diseñador por hora:	11,58€/hora bruto
Sueldo del diseñador:	480 h * 11,58€/h = 5.538€ → 1384€/mes

Por tanto, el coste total del proyecto supondría:

Fabricado y componentes de la tarjeta	-	85,74€
Diseñador/Programador	-	5538€
Software de desarrollo Eagle	-	0€
Software de desarrollo MPLAB IDE	-	0€
TOTAL		5623,74€

También se podría añadir al presupuesto una tarjeta para probar la comunicación can:

CAN Bus Monitor Demo Board	-	49,50€

Y además, incrementar un 5% (ó un 10%) el presupuesto como margen para posibles gastos imprevistos:

(5623,74 + 49,5) * 1,05 = **5957€**

Por lo que el presupuesto requerido para este proyecto tendría un importe equivalente a 5957€

12. PRUEBAS

Las pruebas que se han llevado a cabo han sido:

12.1. Prueba de continuidad en las pistas antes de la soldadura de los elementos
12.2. Prueba visual con microscopio electrónico después de soldar el µC
12.3. Prueba de continuidad después de la soldadura de los elementos
12.4. Prueba de alimentación
12.5. Prueba de funcionamiento de la tarjeta generando un pulso por un pin
12.6. Prueba de lectura de un sensor
12.7. Prueba de lectura de 4 sensores
12.8. Prueba de generación de trama ECAN
12.9. Prueba de recepción de trama ECAN por otro dispositivo

En estas pruebas, se encontraron problemas en los siguientes aspectos:

12.1. Continuidad

Un par de pistas ubicadas en los bordes de la tarjeta fueran cortadas debido al atacado químico tras el insolado. Esto se solucionó uniendo dichas pistas con un poco de estaño y con hilo de *wrapping*.

12.2. Prueba visual con microscopio del µC

Se descubrieron 5 pines cortocircuitados. Esto se solucionó con un soldador de punta fina y flux liquido.

12.3. Prueba de continuidad de los elementos

Se hallaron un par de soldaduras frías. Esto se soluciono aplicando nuevamente el soldador y un poco de estaño.

12.4. Prueba de alimentación

Se percibió un fallo en la alimentación debido a un despiste en la soldadura de un diodo que estaba soldado al revés. Se solucionó soldándolo correctamente.

12.5. Prueba de funcionamiento

Una de las soldaduras frías se había producido en el conector del programador, por lo que no reconocía el dispositivo, tras comprobar continuidad nuevamente se detectó el pin que fallaba y se reparó la soldadura.

12.6. Prueba de lectura de un sensor

Hubo algunos problemas al implementar la función de lectura del sensor, pero con ayuda de un osciloscopio y del *debugger* del Pickit2 se solucionó.

A partir de este punto se presentan los resultados más interesantes obtenidos en el proceso de lectura y pruebas de la aplicación donde la colocación de los dispositivos se muestra en las figuras 12.1, 12.2, 12.3 y 12.4:

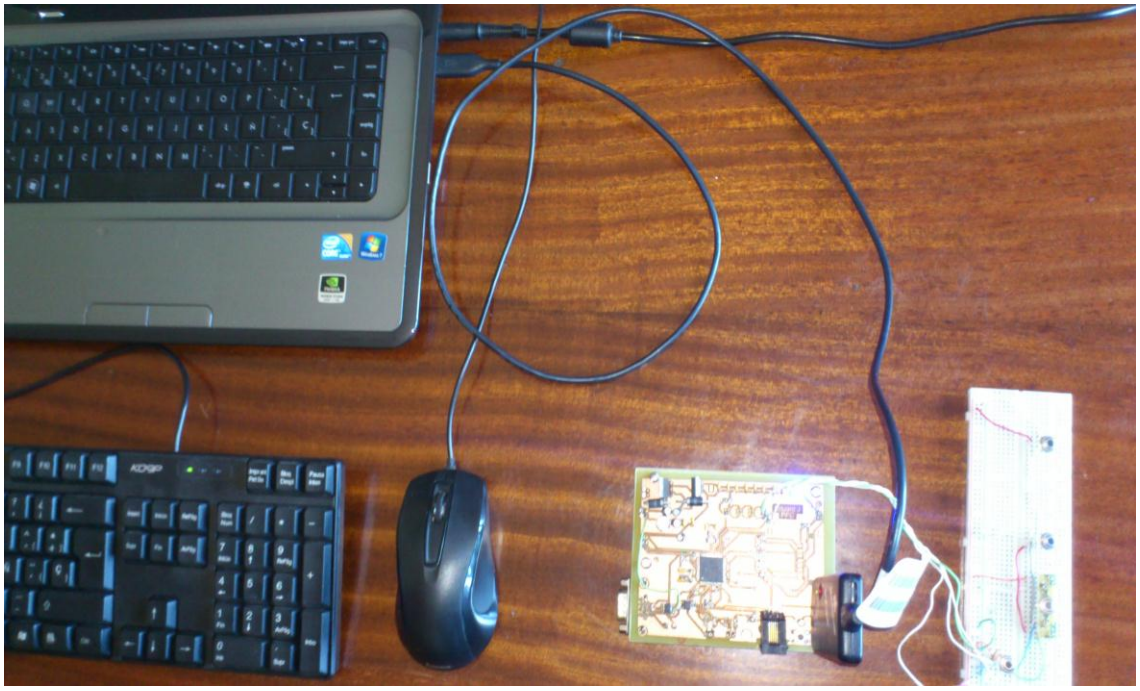


Figura 12.1

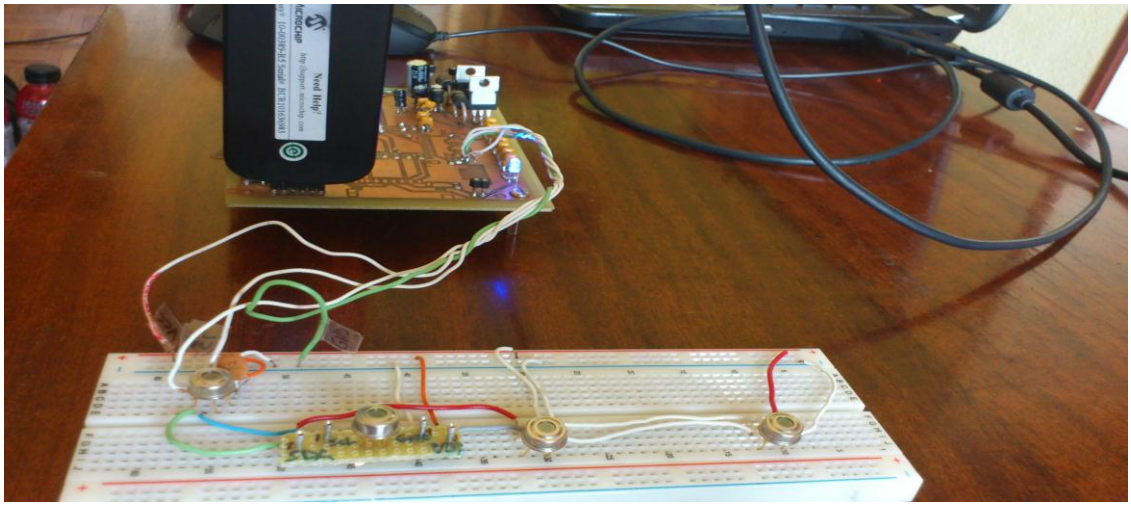


Figura 12.2

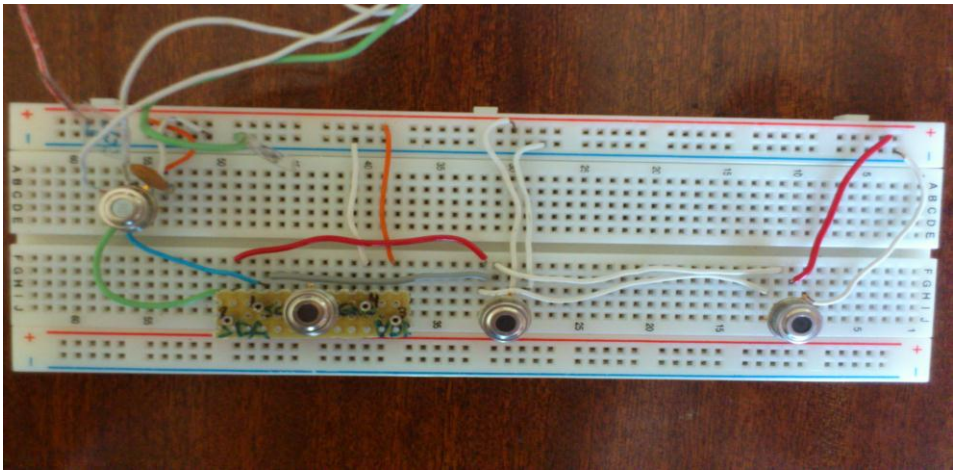


Figura 12.3

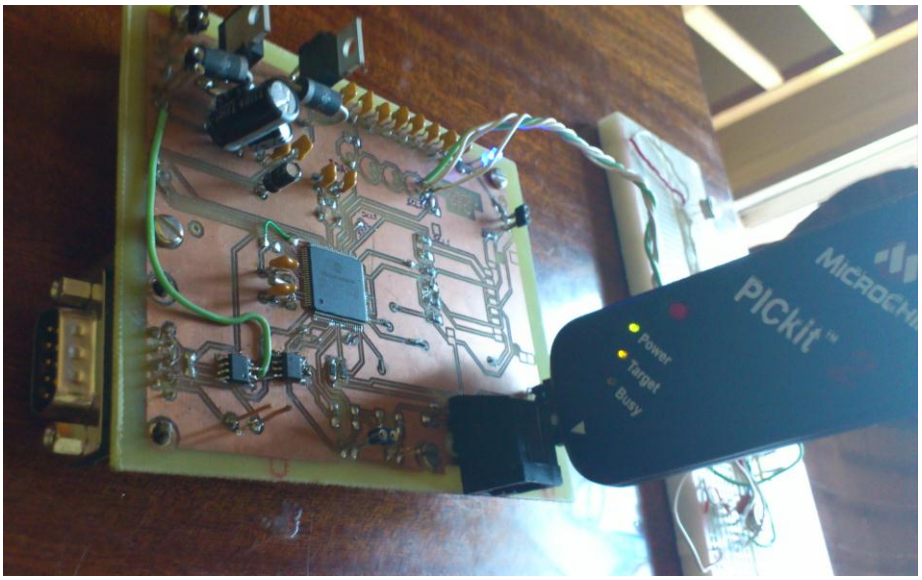


Figura 12.4

12.7. Lectura de los 4 sensores

Para obtener estos resultados, MPLAB IDE proporciona una ventana, *Watch*, que permite recoger los resultados de las variables que gestiona el µC.

En la ventana Watch2 de la figura 12.7.1 se muestran el checksum, los datos enviados por el sensor, el comando de control y la dirección esclava del dispositivo con el que se comunica:

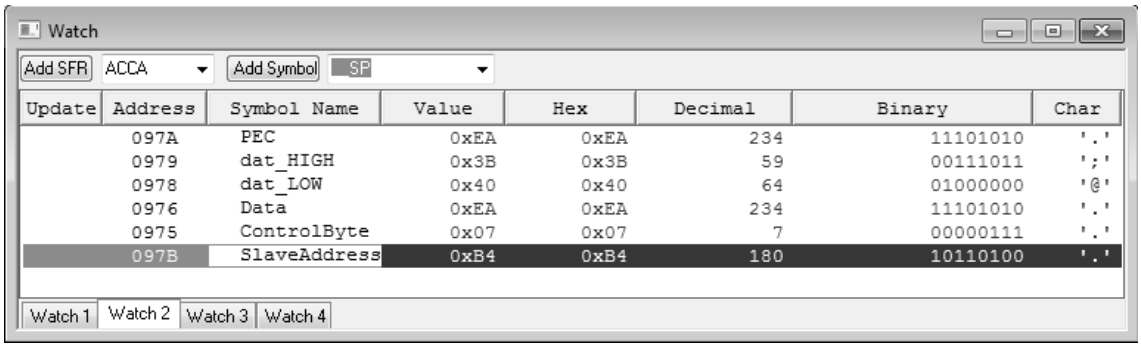


Figura 12.7.1

NOTA: Como estas variables se utilizan de forma recurrente para todos los sensores, sólo es posible mostrar los datos de un único sensor al mismo tiempo. Para ver los 4 sensores simultáneamente recúrrase a las ventanas Watch3 ó Watch4.

En la figura 12.7.2 se observan las temperaturas recogidas del sistema sin que los sensores incidan sobre ningún objeto:

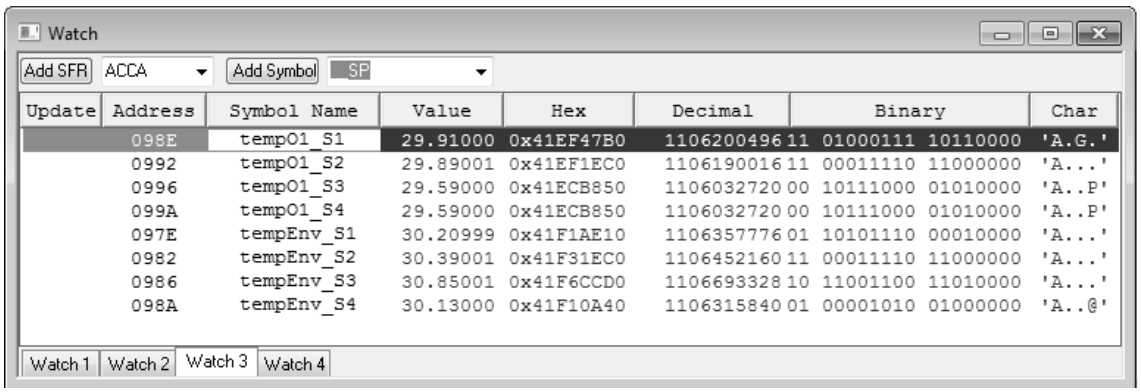


Figura 12.7.2

En la figura 12.7.2 las temperaturas se muestran ya calculadas según la fórmula presentada en la función del código "calc_temperature()", la cual recoge el valor en hexadecimal, lo pasa a decimal, lo divide entre 50 y le resta 273,15.

Véase a continuación un ejemplo del proceso seguido en las capturas, asociadas al sensor de dirección *SlaveAddress* B4:

Dat_HIGH = 0x3B y dat_LOW = 0x40
Luego 3B40 HEX = 15168 DEC
Por tanto 15168 / 50 = 303,36 – 273,15 = 30,2099
30,20999 → Obteniéndose **tempEnv_S1**

dat_HIGH = 0x3B y dat_LOW = 0x31
Luego 3B31 HEX = 15153 DEC
Por tanto 15153 / 50 = 303,06 – 273,15 = 29,91
29,91 → Obteniéndose **tempO1_S1**

Toda la información de la temperatura de los objetos queda registrada en el *arrray* 'sensors' y es posteriormente incorporada a la trama del mensaje ECAN.
La figura 12.7.3 proporciona una visión de la trama ECAN que es transmitida:

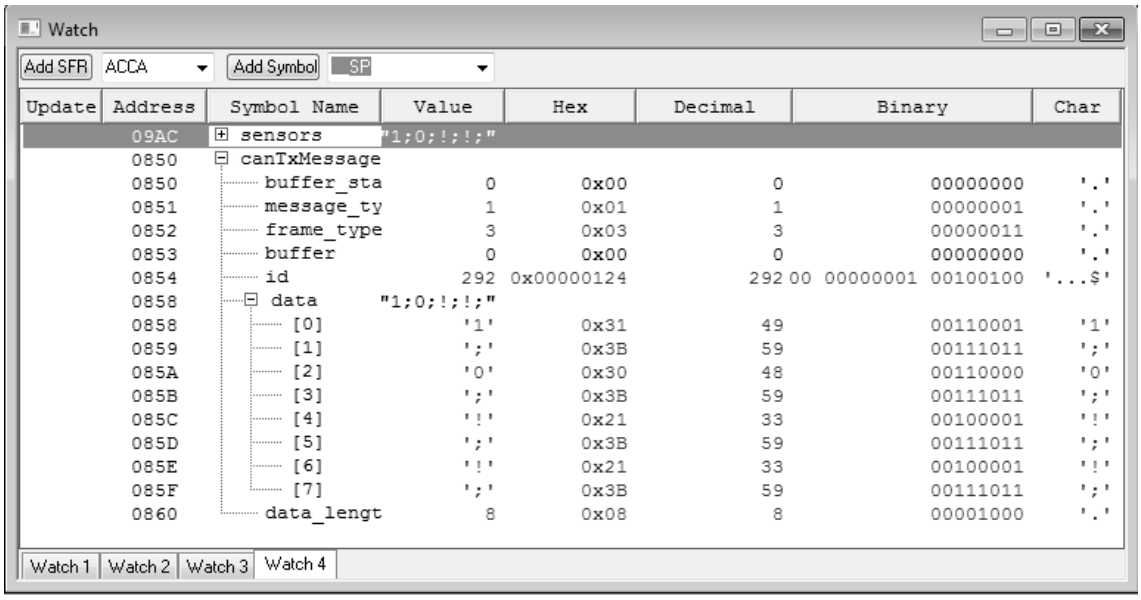
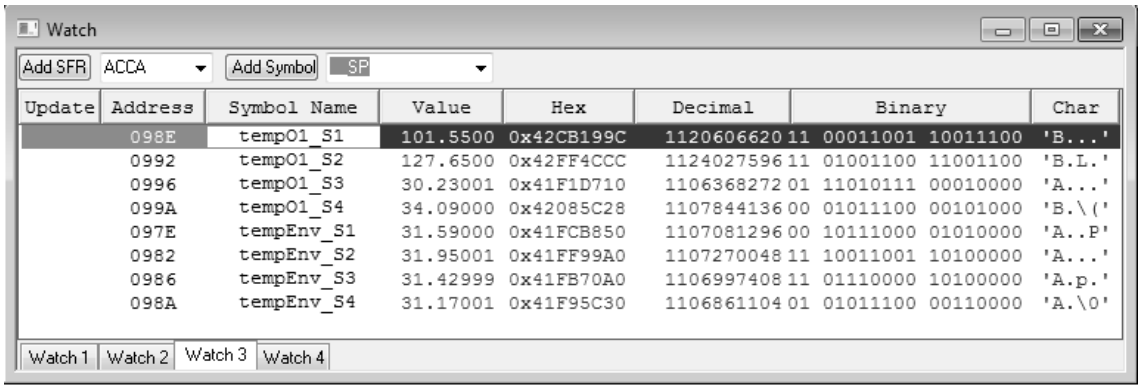


Figura 12.7.3

Para verificar la correcta recogida de datos por parte de los sensores, se presentan distintas capturas tomadas tras hacer incidir su laser infrarrojo sobre distintos objetos:

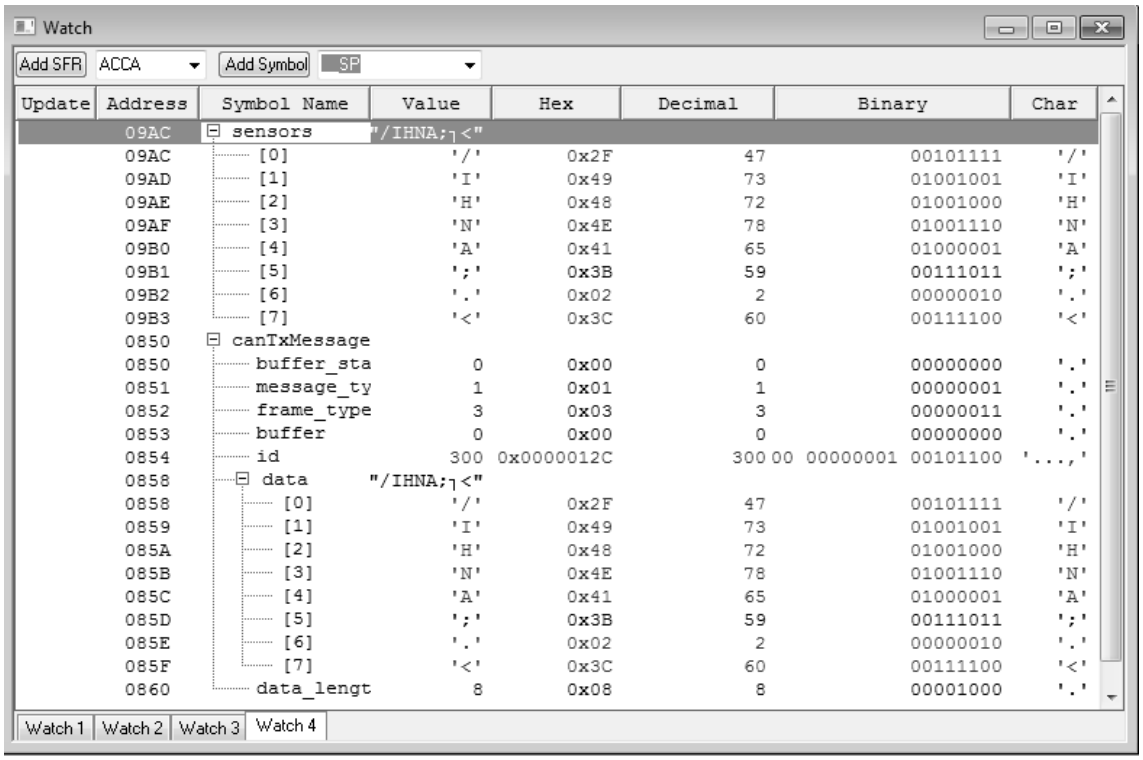
En la figura 12.7.4 se captura la temperatura de un soldador que lleva un par de minutos calentándose y que directamente sólo incide en S1 y S2.



Update	Address	Symbol Name	Value	Hex	Decimal	Binary	Char
	098E	tempO1_S1	101.5500	0x42CB199C	1120606620	11 00011001 10011100	'B...'
	0992	tempO1_S2	127.6500	0x42FF4CCC	1124027596	11 01001100 11001100	'B.L.'
	0996	tempO1_S3	30.23001	0x41F1D710	1106368272	01 11010111 00010000	'A...'
	099A	tempO1_S4	34.09000	0x42085C28	1107844136	00 01011100 00101000	'B.\('
	097E	tempEnv_S1	31.59000	0x41FCB850	1107081296	00 10111000 01010000	'A..P'
	0982	tempEnv_S2	31.95001	0x41FF99A0	1107270048	11 10011001 10100000	'A...'
	0986	tempEnv_S3	31.42999	0x41FB70A0	1106997408	11 01110000 10100000	'A.p.'
	098A	tempEnv_S4	31.17001	0x41F95C30	1106861104	01 01011100 00110000	'A.\0'

Figura 12.7.4

En la figura 12.7.5 se presenta esta última lectura preparada para ser transmitida al bus CAN:



Update	Address	Symbol Name	Value	Hex	Decimal	Binary	Char
	09AC	sensors	"/IHNA;┐<"				
	09AC	[0]	'/'	0x2F	47	00101111	'/'
	09AD	[1]	'I'	0x49	73	01001001	'I'
	09AE	[2]	'H'	0x48	72	01001000	'H'
	09AF	[3]	'N'	0x4E	78	01001110	'N'
	09B0	[4]	'A'	0x41	65	01000001	'A'
	09B1	[5]	';'	0x3B	59	00111011	';'
	09B2	[6]	'.'	0x02	2	00000010	'.'
	09B3	[7]	'<'	0x3C	60	00111100	'<'
	0850	canTxMessage					
	0850	buffer_sta	0	0x00	0	00000000	'.'
	0851	message_ty	1	0x01	1	00000001	'.'
	0852	frame_type	3	0x03	3	00000011	'.'
	0853	buffer	0	0x00	0	00000000	'.'
	0854	id	300	0x0000012C	300 00	00000001 00101100	'...,'
	0858	data	"/IHNA;┐<"				
	0858	[0]	'/'	0x2F	47	00101111	'/'
	0859	[1]	'I'	0x49	73	01001001	'I'
	085A	[2]	'H'	0x48	72	01001000	'H'
	085B	[3]	'N'	0x4E	78	01001110	'N'
	085C	[4]	'A'	0x41	65	01000001	'A'
	085D	[5]	';'	0x3B	59	00111011	';'
	085E	[6]	'.'	0x02	2	00000010	'.'
	085F	[7]	'<'	0x3C	60	00111100	'<'
	0860	data_lengt	8	0x08	8	00001000	'.'

Figura 12.7.5

En la figura 12.7.6 se captura la temperatura de diversos congelados:

Watch

Add SFR | ACCA | Add Symbol | SP

Update	Address	Symbol Name	Value	Hex	Decimal	Binary	Char
	098E	tempO1_S1	-6.470001	0xC0CF0A40	3234794048	11 00001010 01000000	'...@'
	0992	tempO1_S2	-12.79001	0xC14CA3E0	3243025376	00 10100011 11100000	'.L..'
	0996	tempO1_S3	-17.92999	0xC18F70A0	3247403168	11 01110000 10100000	'..p.'
	099A	tempO1_S4	-6.889984	0xC0DC7AC0	3235674816	00 01111010 11000000	'..z.'
	097E	tempEnv_S1	30.64999	0x41F53330	1106588464	01 00110011 00110000	'A.30'
	0982	tempEnv_S2	30.83002	0x41F6A3E0	1106682848	10 10100011 11100000	'A...'
	0986	tempEnv_S3	30.59000	0x41F4B850	1106557008	00 10111000 01010000	'A..P'
	098A	tempEnv_S4	30.63000	0x41F50A40	1106577984	01 00001010 01000000	'A..@'

Watch 1 | Watch 2 | Watch 3 | Watch 4

Figura 12.7.6

12.9. Recepción de trama ECAN por otro dispositivo

Para la recepción de tramas ECAN se propone usar cualquier dispositivo que disponga de periféricos adecuados, pero en concreto se recomienda emplear la tarjeta CAN Bus Monitor Demo Board de Microchip debido a su completa y sencilla guía de usuario.
En la figura 12.9.1 se muestra este tipo de dispositivos:

Figura 12.9.1

Estos dispositivos van conectados a un ordenador mediante USB y controladas por un software específico y fácil de configurar. En la figura 12.9.2 se muestra su panel de control/configuración:

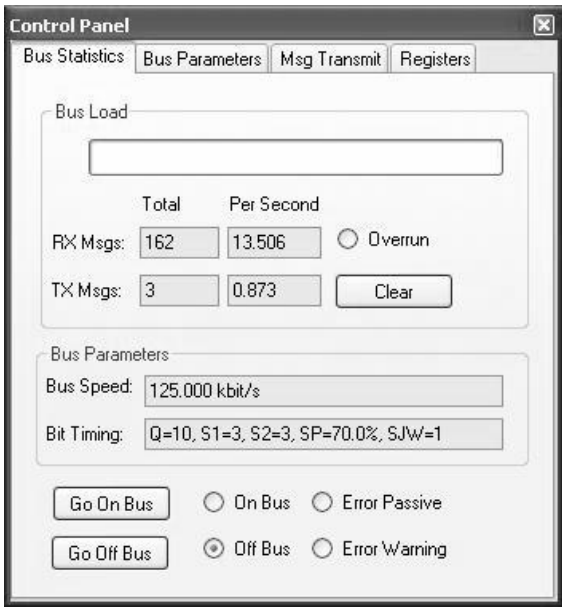


Figura 12.9.2

Los resultados, como se muestra en la figura 12.9.3, se recogen en una ventana exclusiva para los mensajes CAN:

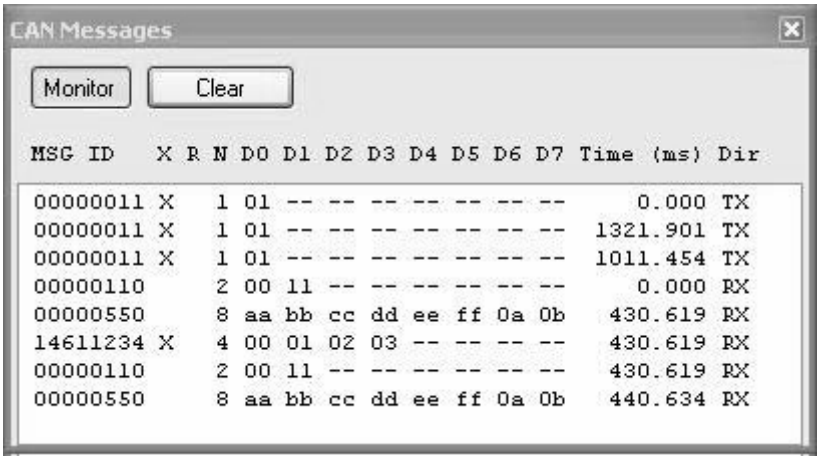


Figura 12.9.3

También se recomienda usar la tarjeta Explorer 16 Development Board, si se dispone de acceso a ella, para interactuar con nuestro dispositivo mediante el bus CAN, incluso se recomienda adquirir dicha tarjeta ya que sus prestaciones son altas y su precio no es demasiado elevado: 149,99$ → 105€

En la figura 12.9.4 se muestra esta tarjeta:

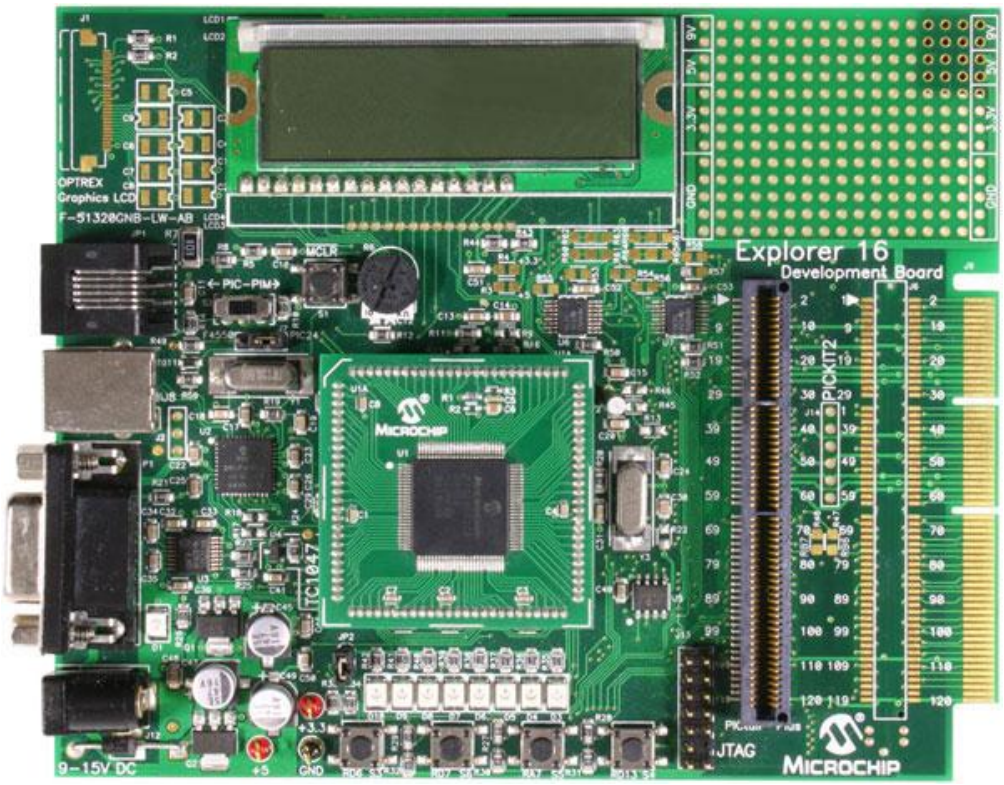


Figura 12.9.4

La ventaja de esta tarjeta, es que además de poder mostrar los datos en un ordenador, dispone de su propio LCD para mostrar las capturas y resultados deseados en el propio dispositivo.

13. CONCLUSIONES

Conclusiones sobre el proyecto

Ya que este proyecto pretende el estudio de la evolución en la temperatura de los neumáticos para de ese modo optimizar los recursos y mejorar la gestión de los mismos, se puede destacar este proyecto como una aplicación útil también para los fabricantes de neumáticos que podrían utilizar los datos para mejorar sus productos.

Conclusiones sobre el diseño

La solución propuesta ha sido una tarjeta que responde a los requisitos, sin embargo, una vez finalizado el desarrollo se detecta lo siguiente:

- El tamaño final de la tarjeta podría haber sido una cuarta parte del tamaño resultante, es decir: 2,5cm de ancho x 10 cm de largo.
- Se podrían haber puesto todos los elementos en encapsulado SMD, lo cual reduciría la altura de la tarjeta, pero aumentando su coste.
- La frecuencia a la que trabaja la tarjeta, 80 MHz, es sobradamente alta, por lo que incluso con 20MHz funcionaría correctamente y consumiría menos.
- Como tarjeta prototipo de desarrollo que se incluya la posibilidad de que sea programada por ICD2 y Pickit2 mejora sus capacidades, pero es innecesario para su utilidad.
- Para desarrollar el código se podría haber utilizado la tarjeta de Microchip Exporer16.
- Se podría haber incluido una pantalla LCD para la visualización de los datos y la depuración de los códigos que se van desarrollando en la tarjeta, aunque esto aumentase el coste total de la misma y dificultase tanto el ruteado de las pistas como el desarrollo del código.

13.1. MEJORAS y USOS DE LA TARJETA

Mejoras propuestas

- Se propone incluir la tarjeta SD para capturar los datos recogidos con un sistema de archivos de formato '.txt'.
- Se propone incluir un pulsador ó unos micro-interruptores para ampliar las posibilidades ofrecidas por la tarjeta.
- Se propone incluir en el código un freeRTO (Sistema Operativo de Tiempo Real open source) para la gestión por tareas de las funciones.

Usos adicionales de esta tarjeta

Entre las muchas posibilidades que ofrece la tarjeta implementada, se dispone para el desarrollo de aplicaciones los siguientes elementos ya incluidos:

- Dos entradas de interrupción para la captura de datos, señales, pulsadores, o cualquier otro elemento que se pueda agregar.
- Dos entradas ADC para la captura de datos de señales, baterías, monitorización, o cualquier otra gestión de recursos o sensores que se requiera.
- Acceso a un bus ECAN.
- Acceso a un bus I2C.
- Acceso a una memoria flash.
- Acceso a una memoria SD.
- Salida de tensión de 5V y de 3,3V.

Con estos elementos se abren multitud de posibilidades, quedando a expensas su uso tanto de los requisitos de una aplicación, así como de la imaginación del programador y de las distintas combinaciones de los elementos disponibles.

13. BIBLIOGRAFÍA

Documentación sobre Formula SAE:

URL: http://students.sae.org/competitions/formulaseries/

Documentación del µControlador y sus elementos internos:

URL: http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en024679

Documentación sobre los integrados empleados en el diseño:

M25P80

URL: http://www.datasheetcatalog.org/datasheet/stmicroelectronics/8495.pdf

LM1117T

URL:

http://www.datasheetcatalog.org/datasheet2/8/0uzr32fcc5ych0637cho90kh7f3y.pdf

MCP2551

URL: http://ww1.microchip.com/downloads/en/devicedoc/21667d.pdf

Documentación sobre tarjetas SD:

URL: http://es.wikipedia.org/wiki/Secure_Digital

Documentación sobre el sensor MLX90614:

URL: http://www.melexis.com/Infrared-Thermometer-Sensors/Infrared-Thermometer-Sensors/MLX90614-615.aspx

URL: http://www.melexis.com/Forums/Sensor-ICs-Infrared-Thermometers-21.aspx

Documentación sobre conectores (*headers*):

URL: http://www.surplussales.com/computeraccess/IHJConnectors.html

Documentación sobre Eagle:

URL: http://www.cadsoftusa.com/

URL: http://www.cadsoftusa.com/downloads/libraries?language=en

Información sobre protocolo ECAN:

URL: http://www.interfacebus.com/CAN-Bus-Description-Vendors-Canbus-Protocol.html

URL: http://www.semiconductors.bosch.de/en/ipmodules/can/can.asp

Información sobre protocolo I2C:

URL: http://en.wikipedia.org/wiki/I%C2%B2C

URL: http://labjack.com/support/app-notes/mlx90614-ir-temperature-sensor-i2c

Documentación auxiliar sobre la tarjeta Explorer16 de Microchip:

URL: http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en024858&part=DM240001

Documentación auxiliar sobre la tarjeta de monitorización de bus CAN:

URL:

http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en537141&part=MCP2515DM-BM

14. MANUAL DE USUARIO

Para un empleo correcto de la tarjeta, se añade un breve manual de usuario para facilitar al usuario un rápido conocimiento de la tarjeta implementada.

En la figura 14.1 se enumeran las partes que componen la tarjeta para su posterior descripción detallada:

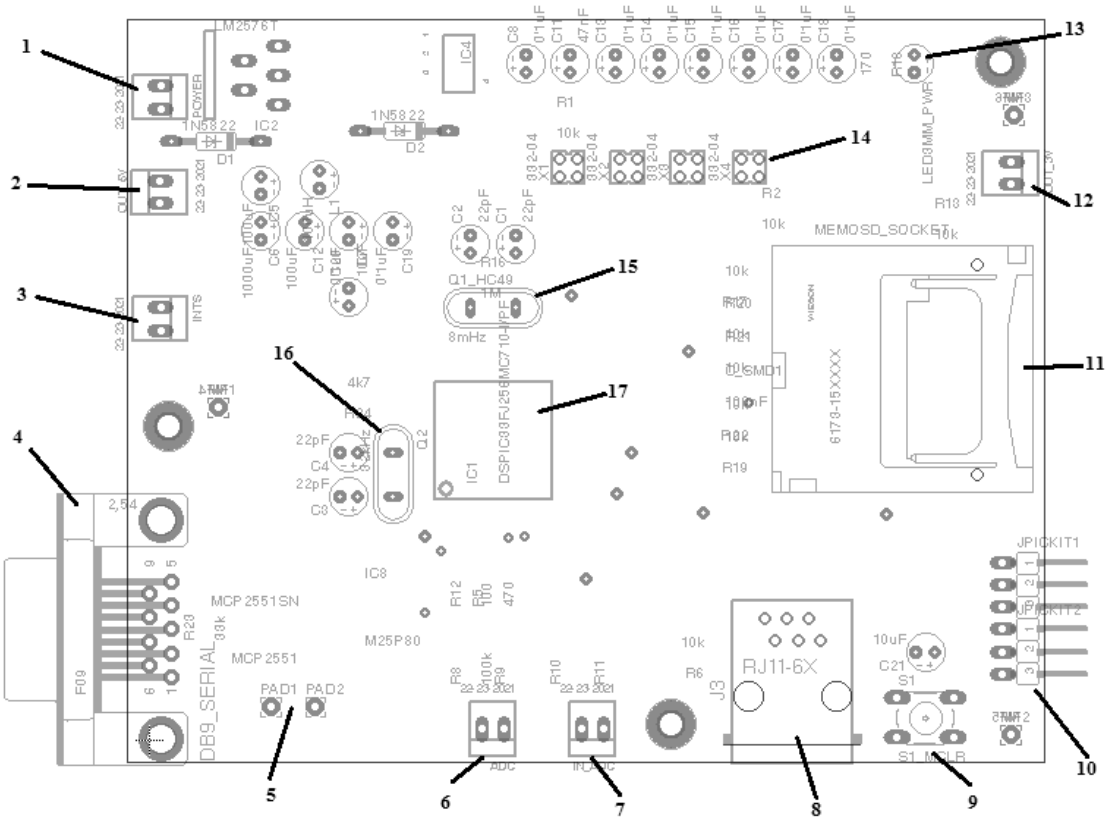


Figura 14.1

1. Entrada de alimentación	**10**. Conector para programador Pickit2
2. Salida de 5V	**11**. Socket para tarjeta SD
3. Dos entradas de interrupción	**12**. Salida 3,3V
4. Conector entrada/salida ECAN	**13**. LED indicador de alimentación 3,3V
5. Pines de test ECAN	**14**. Conectores de bus I2C
6. Entrada ADC ó salida digital	**15**. Oscilador primario (8MHz)
7. Entrada ADC ó salida digital	**16**. Oscilador secundario (32KHz)
8. Conector para programador ICD2	**17**. µC dsPic33FJ256GP710
9. Botón de reset global	

A continuación se detallan aquellos elementos que requieren un cuidado especial a la hora de conectarlos debido a su polaridad o conexionado en la PCB.

1. Entrada de alimentación

La alimentación debe estar comprendida entre 9V y 30V para un correcto funcionamiento. La polaridad se muestra en la figura 14.2:

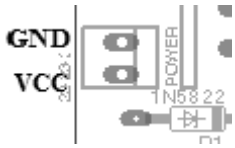


Figura 14.2

2. Salida de 5V

La polaridad para obtener los 5V es la mostrada en la figura 14.3:

Figura 14.3

10. Conector para programador Pickit2

Es necesario saber la posición de conexión adecuada del programador, por lo que se debe conocer cual es el pin 1, el cual esta señalado en la figura 14.4:

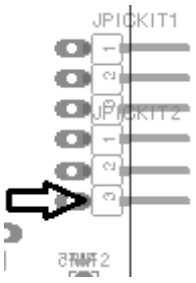


Figura 14.4

12. Salida de 3,3V

En la figura 14.5 se aprecia la polaridad de este conector:

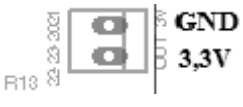


Figura 14.5

14. Conector de bus I2C

El bus I2C compartido por los conectores cuadrados esta implementado como se muestra en la figura 10.6:

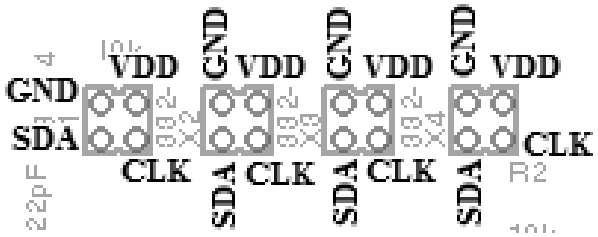


Figura 10.6

NOTA: A este bus se puede conectar cualquier dispositivo que funcione a 3,3V.