Proyecto Final
Robot recolector de residuos
Diseño, implementación y construcción física

Guillermo Campelo
Juan Ignacio Goñi
Diego Nul

Tutor: Prof. Juan Manuel Santos

Instituto Tecnológico de Buenos Aires
Departamento de Informática

31 de julio de 2010

Resumen

Palabras clave: *Robot, recolector, residuos, robótica, comportamientos, visión, MR-2FA, L298, FR304, HX5010, GP2D120, BC327, SRF05, CNY70, motor, servo, telémetro, daisy chain, RS232, subsumption, Webots, OpenCV, detección, objetos*

Índice

Índice de cuadros

Índice de figuras

1. Requerimientos

2. Ideas de implementación

2.1. Locomoción

distintos tipos de locomocion que tuvimos en cuenta y porque elegimos este

2.2. Sensado del entorno

distintos tipos de sensores disponibles y porque elegimos estos

2.3. Controlador

distintas formas de diagramar la forma de control, que tipo de controladores necesitamos, en cuales pensamos, con cuales nos quedamos

2.4. Método de recolección

distintos metodos que se nos ocurrieron

3. Actuadores

La principal forma en que el robot puede interactuar activamente con el ambiente que lo rodea son los motores y cada tarea que debíamos realizar requería de actuadores acordes. Estas cuestiones son las que analizamos en este apartado.

3.1. Motores de contínua

Para la tracción principal de las ruedas necesitabamos motores que tuvieran el torque suficiente para mover el robot, pero que pudieramos medir y controlar la velocidad era la principal necesidad. Para esta tarea utilizamos motores de contínua con caja reductora y encoder. Con dos de estos motores logramos poder garantizar una velocidad determinada en las ruedas, controlar de la cantidad de movimiento en forma independiente en cada rueda y entre otras cosas conocer la cantidad de las vueltas dadas por cada una de las ruedas.

3.1.1. Características

Los motores que elegimos son de la marca Ignis[1] modelo *MR-2FA* con características expresadas en el cuadro 23, están provistos de una caja reductora, poseen un encoder de 4 estados por vuelta en el eje del motor y un sensor de de efecto de campo para determinar una vuelta en la salida de la caja reductora. En la figura 1 mostramos las dimensiones exteriores del motor.

Una ventaja que encontramos en este modelo es que ya trae el encoder integrado aunque su resolución podría haber sido mayor. Los encoders los explicamos más en detalle en la sección 4.4. La caja reductora provee una relación de 94 vueltas del motor por cada 1 vuelta del eje de salida de la caja.

[1] http://www.ignis.com.ar

Característica	Unidad	Mínimo	Nominal	Máximo
Tensión	V	8	9	12
Corriente	A	0.6	1.2	2.4
Velocidad	RPM	1	60	60
Aceleración	$1/s^2$	0.1	0.1	0.5
Torque	kgf*cm	0	1.2	6.4

Cuadro 1: Características del motor Ignis MR-2FA.

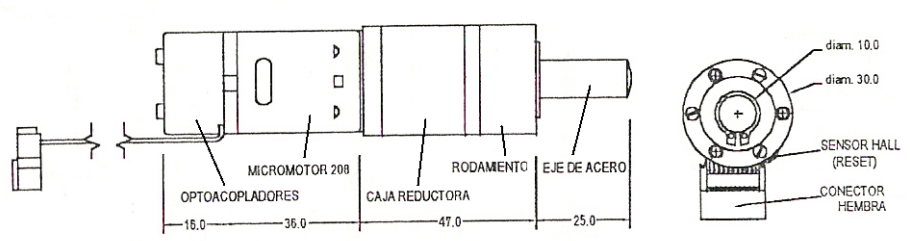

Figura 1: Vista lateral y frontal del motor Ignis MR-2FA.

3.1.2. Circuito de control

Para alimentar y poder controlar los motores elegimos el driver *L298* de la marca ST[2]. Internamente tiene dos puentes H puenteables y puede soportar hasta $4A$. Esto lo logramos teniendo las salidas 1 y 2 están puenteadas con las salidas 4 y 3 respectivamente, como mostramos en el diagrama de la figura 24. Ideal para estos motores u otros que se elijan en el futuro.

La principal función del driver era proveer de la corriente y voltaje necesarios para el funcionamiento del motor, pero la configuración del puente H nos dio la posibilidad de, con una lógica simple, determinar el sentido de la corriente y potencia que recibía el motor.

Este integrado admite el puenteo de las salidas aumentando así la corriente que circulará por el motor. Para hacer esto, conectamos las salidas *Out1* y *Out4* por un lado y por el otro las salidas *Out2* y *Out3*. De igual forma los pines de habilitación *EnA* y *EnB*, luego la entrada *In1* con la *In4* por un lado y por el otro la *In2* con la *In3*.

De esta forma, se controla con sólo 3 cables, uno de habilitación y otros 2 de *Input* que determinan la polarización de los transistores internos y por ende, el sentido de giro del motor. En el cuadro 2 mostramos la tabla de verdad para los pines de control, donde H es estado alto, L estado bajo y X cualquier estado. En la figura 2 mostramos el diagrama interno del integrado.

Para determinar la potencia que recibirá el motor usamos el módulo de *PWM* del microcontrolador, que explicamos más en detalle en la sección 5.2.2. Variando el ancho del pulso sobre el pin de habilitación del driver determinamos la cantidad de tiempo que el motor recibe tensión, lo cual se traduce en la potencia que este tiene para realizar el movimiento. Cuanto mayor es el tiempo en estado alto del pulso, mayor la potencia.

[2]http://www.st.com

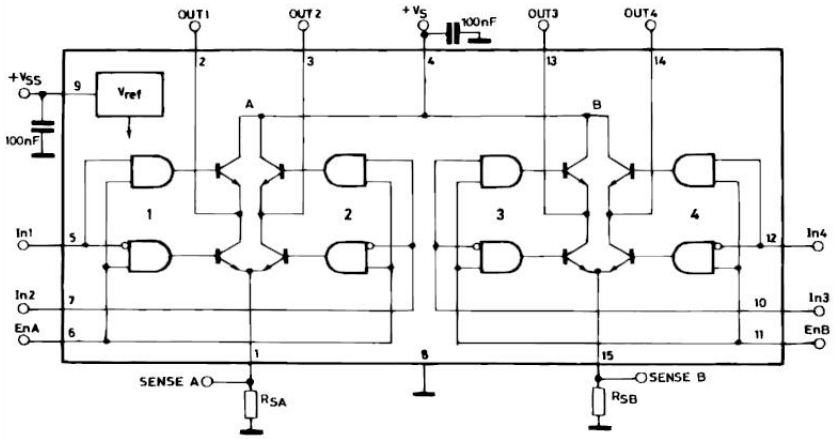

Figura 2: Diagrama interno del driver L298.

Enable	Input 1	Input 2	Función
H	H	L	Sentido horario
H	L	H	Sentido anti-horario
H	L	L	Motor frenado
H	H	H	Motor frenado
L	X	X	Motor libre

Cuadro 2: Tabla de verdad para el control del driver *L298*.

Para contrarrestar la corriente negativa en las salidas del driver usamos los diodos *FR304* que cumplen con las especificaciones del driver con 150ns de tiempo de recuperación y una corriente de $3A$.

El consumo del motor lo medimos mediante los pines de sensado en el driver, conectado a masa por una resistencia y al módulo de *ADC* del microcontrolador, que explicamos en la sección 5.2.2. Conociendo el valor de la resistencia y el valor leido por el módulo de *ADC*, pudimos determinar cuanta corriente que circulaba por la resistencia y por ende la corriente consumida por el motor.

Para controlar la velocidad del motor usamos uno de los dos encoders que trae. Conectados como entrada para el módulo de *Timer* del microcontrolador, en configuración de contador, que explicamos más en detalle en la sección 5.2.2. Usando otro *Timer* para tener una base de tiempo fija y con el valor del contador pudimos determinar y controlar la velocidad de las ruedas.

En la sección 7.2 explicamos el desarrollo de la placa controladora de estos motores.

3.1.3. Rutinas de control

Desde el punto de vista del código, tuvimos que desarrollar las rutinas necesarias para el manejo de los motores según las intrucciones del controlador principal.

Configuramos a uno de los timers internos del microcontrolador para que genere una interrupción cada 6,25ms, la cual usamos para realizar chequeos y

Característica	Unidad	Valor
Torque	kg	6,5
Velocidad	segundos/grado	$\frac{0,16}{60}$
Voltaje	V	4,8 a 6
Delay máximo	μs	4
Dimensiones	mm	40x20x38
Peso	g	39

Cuadro 3: Características del servo HX5010.

ejercer control sobre el motor. Verificamos el consumo del motor para evitar sobrecargar el circuito y los motores ante un posible atasco de las ruedas. También actualizamos el acumulado de vueltas realizadas por el motor para la odometría.

Cada 200ms (32 interrupciones) tomamos la cantidad de cuentas del encoder, corregimos la velocidad de giro del motor ajustando el ancho del pulso generado por el PWM. Luego borramos el contador y esperamos otros 200ms.

3.2. Servo motores

Para el movimiento de las partes del módulo de recolección, una cámara con paneo y giro o un senor de ultrasonido colocado en la parte superior haciendo las veces de radar, pensamos en el uso de servo motores. La principal característica de estos actuadores es que sólo debemos indicar el ángulo al que queremos que este el eje del motor y este se coloca automáticamente. El ángulo de trabajo va desde 0° a 180° y algunos llegan hasta los 200°.

Aunque no fue implementado el ningún mecanismo que requiriera el uso de estos motores, explicamos en este apartado el trabajo realizado en torno a este tipo de actuadores. En la sección 7.4 explicamos el diseño de las placas que los controlan.

El servo de prueba que utilizamos para el desarrollo es el modelo *HX5010* de la marca Hextronik[3] con caracteristicas que expresamos en el cuadro 3

3.2.1. Circuito de control

La alimentación y consumo depende del modelo específico, variando también el torque que posee el servo.

No es necesario el uso de un driver para manejarlos, simplemente con la alimentación y una señal con el ángulo es suficiente. La forma de comunicar el ángulo varía entre los distintos servos y fabricantes. Hay servos analógicos y servos digitales. En los primeros la posición se determina mediante un voltaje que varia según cierto rango y si es digital, se setea mediante el ancho de un pulso que tiene un tiempo mínimo y máximo para mapear los ángulos mínimo y máximo respectivamente.

Dentro del modo de uso, podemos hacer que queden sueltos o que se queden fijos en cierta posición indicando, de forma contínua, el valor del ángulo requerido. La frecuencia a la que se debe setear la posición depende el modelo.

[3] http://www.hextronik.com/

3.2.2. Rutinas de control

Debido a que pensamos usar servos digitales y por lo menos íbamos a necesitar 3 servos, necesitabamos contar con varios módulos de PWM. Como sólo disponíamos de 1, decidimos realizar la misma función pero por software.

Usamos el timer de 16bits del microcontrolador configurado con el clock interno como medida del tiempo para crear 5 salidas con pulsos que varían su ancho en forma independiente cada una. Definimos un ancho mínimo y máximo, pudiendo configurar pasos intermedios de 1° (aproximadamente 69,4μs).

4. Sensado

En este apartado explicamos detalladamente cada uno de los sensores que utilizamos para realizar tanto las mediciones externas como las internas al robot. Analizamos las ventajas de cada uno, problemas que encontramos y sus soluciones.

En la sección 7.3 explicamos el diseño y construcción de las placas que controlan todos los sensores del robot.

4.1. Telémetros infrarrojos

El principio de funcionamiento de estos sensores es mediante un haz de luz infrarroja que es emitido hacia el objetivo, el cual es reflejado y captado a traves de un lente por un sensor de posición relativa en el interior del sensor. En base a esta medición se calcula la distancia entre el sensor y el objeto reflectivo que se encuentra frente a él.

4.1.1. Características

Los telémetros infrarrojos que elegimos son de la marca Sharp[4], modelo *GP2D120*. En el cuadro 4 detallamos los valores característicos del modelo.

Este tipo de sensores tienen un retardo de aproximadamente 43,1ms durante el cual la lectura que se realiza no es confiable y luego las nuevas lecturas se hacen en ventanas de aproximadamente el mismo tiempo. En la figura 3 mostramos el diagrama de tiempos.

Característica	Unidad	Valor
Rango máximo	cm	30
Rango mínimo	cm	4
Tensión para la máxima distancia	V	1.95
Tensión para la mínima distancia	V	2.55
Tensión de alimentación	V	5
Consumo máximo	mA	50

Cuadro 4: Características del sensor de distancia por ultrasonido SRF05.

[4]http://sharp-world.com/products/device

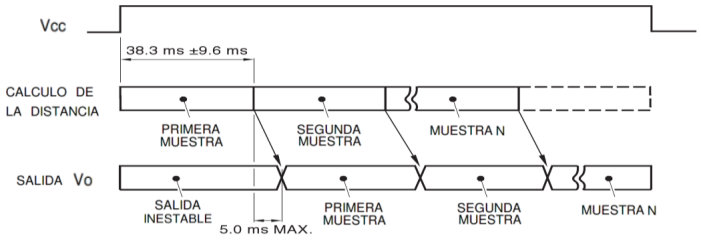

Figura 3: Diagrama de tiempos del sensor GP2D120.

4.1.2. Circuito de control

No necesitábamos un driver para manejar los telémetros, pero decidimos usar un transistor para poder habilitarlos o no, de otra manera el sensor estaba tomando mediciones continuamente provocando un consumo de batería innecesario. De esta forma sólo se enciende cuando se va a utilizar.

El transistor que utilizamos es un conmutador y amplificador de uso general, el *BC327* y por ser de tipo PNP se exita con un estado bajo, por lo que la lógica de conmutación está negada.

El tipo de salida de este modelo de telémetros es analógica y están conectadas al módulo *ADC* del microcontrolador. En la figura 4 mostramos el cuadro de conversión entre voltaje de salida y distancia al objeto, y en la figura 5 mostramos el ángulo de apertura de la zona de detección según la distancia al objetivo.

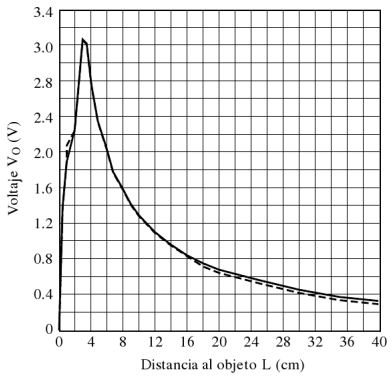

Figura 4: Voltaje de salida según la distancia al objeto del telémetro GP2D120.

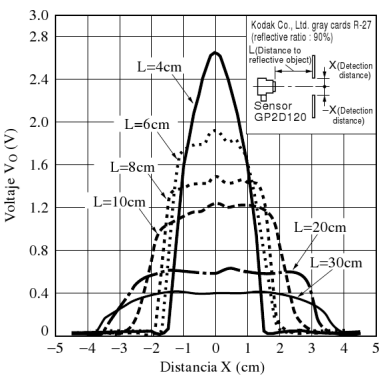

Figura 5: Ángulo de apertura según la distancia del telémetro GP2D120.

4.1.3. Rutinas de control

Controlar los telémetros es relativamente sencillo. Usamos el timer de 16bits del microcontrolador configurado con el clock interno para determinar el tiempo en el cual debíamos tomar las muestras con el ADC y simplemente realizamos un promedio entre ellas para obtener la distancia al objetivo.

Para hacer esto desarrollamos una pequeña máquina de estados que controla

y maneja los tiempos para tomar las muestras que explicamos más en detalle en la sección 7.3.7.

4.2. Sensor de distancia por ultrasonido

Estos sensores de distancia se basan en la velocidad del sonido para calcular la distancia al objetivo. Genera un tren de 8 pulsos ultrasónicos y luego se espera como respuesta, el mismo tren de pulsos que debería haber rebotado contra el objetivo. En base a la diferencia de tiempo entre la emisión del tren de pulsos y la respuesta, se calcula la distancia a la que se encuentra el objetivo.

4.2.1. Características

El sensor de distancia por ultrasonido que elegimos es el modelo *SRF05* de la marca Devantech Ltd[5]. Esta versión mejorada del modelo *SRF04*, aumenta el rango de detección y mejora el modo de control y lectura de los datos, permitiendo hacerlo mediante un único pin.

La distancia medida mediante el tren de pulsos es codificada linealmente en el ancho de un pulso que varía de $100\mu s$ a $25ms$. Si dentro del rango de detección no se encuentra ningún objeto, el pulso tendrá un ancho de $30ms$.

En el cuadro 5 detallamos las caracterśticas del sensor *SRF05* y en la figura 6 mostramos el haz ultrasónico del sensor.

Característica	Unidad	Valor
Tensión de alimentación	V	5
Corriente	mA	4
Frecuencia de trabajo	KHz	40
Rango máximo	cm	400
Rango mínimo	cm	1.7
Duración mínima del pulso de disparo	μs	10
Duración del pulso eco de salida	μs	100 - 25000
Tiempo mínimo de espera entre mediciones	ms	50
Dimensiones	mm	43x23x40

Cuadro 5: Características del sensor SRF05.

4.2.2. Circuito de control

No necesitamos de un driver para manejar al sensor ya que lo conectamos directo a los $5v$ de la placa. El pin de *Mode* lo dejamos en estado bajo para indicar que debe funcionar bajo el nuevo modo y no en compatibilidad con el *SRF04*.

En el pin de *TRIGGER* sólo generamos un pulso de al menos $10\mu s$ para desencadenar en la lectura de la distancia al objetivo. El sensor nos asegura que no generará el pulso de respuesta hasta pasados los $700\mu s$ desde pasado el pulso de trigger. En la figura 7 mostramos el diagrama de tiempos.

Para evitar que el rebote de otros sensores o sensados anteriores influya en la lectura, se debe esperar un mínimo de $50ms$ antes de generar otro medición.

[5] http://www.robot-electronics.co.uk/

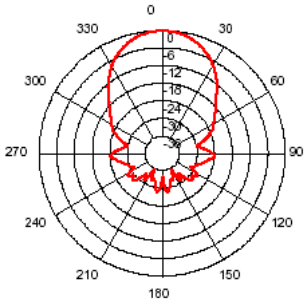

Figura 6: Haz ultrasónico del sensor SRF05.

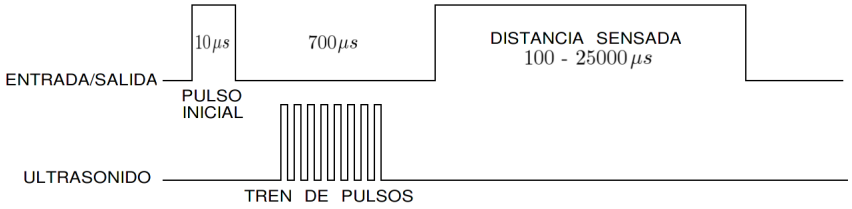

Figura 7: Diagrama de tiempos del sensor SRF05.

4.2.3. Rutinas de control

Usamos uno de los pines con interrupción externa en el que conectamos el pin de *TRIGGER* y el timer de 16bits del microcontrolador configurado con el clock interno.

Para realizar la medición, generamos un pulso de $15\mu s$ para asegurarnos el disparo del sensor y cambiamos el modo del pin a entrada con interrupción ante un flanco ascendente. Cuando salta la interrupción significa que comienza al pulso con la distancia codificada en su ancho, por lo que tomamos una muestra del timer y configuramos al pin para que genere ahora una interrupción ante un flanco descendente. Cuando salte la próxima vez la interrupción, será porque termino el pulso con la medición, por lo que sólo debemos hacer la resta entre el valor actual del timer y la muestra que tomamos al principio para conocer la distancia a la que se encuentra el objetivo.

Un tiempo obtenido mayor a los $25ms$ indica que no se detectó ningún objeto dentro del rango del sensor.

4.3. Sensor reflectivo de piso

Estos sensores opticos reflectivos emiten luz infrarroja y captan el nivel de luz reflejada sobre la superfície a sensar como muestramos en la figura 9. En la figura 8 mostramos las dimensiones del sensor. La intensidad de luz captada depende de la distancia al objetivo y del color y nivel de reflectividad de la superficie. Es por esto que usamos estos sensores para identificar una línea en el piso.

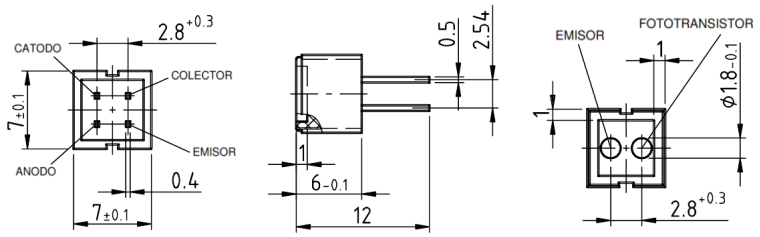

Figura 8: Medidas en milímetros del sensor CNY70.

4.3.1. Características

Los sensores que elegimos para son del modelo *CNY70* de la marca Vishay Semiconductor[6].

El rango efectivo de sensado ronda los $3mm$ de distancia aunque, con un incremento en la corriente que circula por el emisor se puede llegar a una distancia mayor a la recomendada por el fabricante y que nos permita un uso más acorde al proyecto. El emisor soporta un pulso de hasta $3A$ por un tiempo menor o igual a $10\mu s$.

En la figura 10 mostramos la corriente que circula por el colector del fototransistor en base a la distancia al objeto medido.

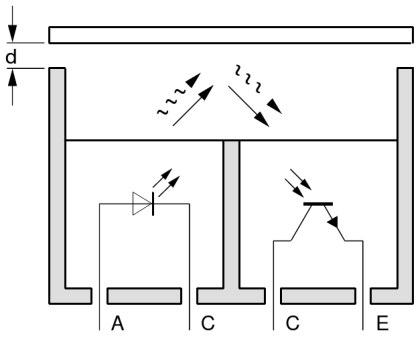

Figura 9: Principio de funcionamiento reflectivo del sensor CNY70.

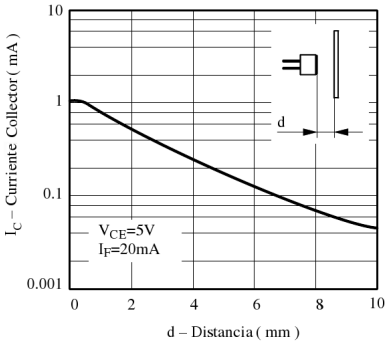

Figura 10: Corriente en el colector según la distancia del sensor CNY70.

4.3.2. Circuito de control

De igual forma que en los telémetros utilizamos un transistor como habilitación de la tensión de alimentación en el sensor. Esto nos dio la posibilidad de encenderlo y apagarlo a la hora de tomar las muestras de la luz reflejada por piso o la línea. Nuevamente la lógica de habilitación es invertida por tratarse de un transistor *BC327*.

[6]http://www.vishay.com/

Agragamos una resistencia para limitar la corriente que circulaba por el emisor y otra como pull-up en el emisor del fototransistor que a su vez, conectamos al módulo de *ADC* del microcontrolador para efectuar las mediciones.

4.3.3. Rutinas de control

El código que desarrollamos para obtener las muestras de estos sensores es sencillo, simplemente debemos habilitar el transistor que alimenta al sensor con un estado lógico bajo, tomar al menos 4 muestras y promediarlas para tener un valor adecuado del nivel de luz reflejado por la superficie. Deshabilitamos el sensor y enviamos el valor.

Debido al circuito que armamos con un nivel alto de reflexión leemos un valor bajo en el conversor analógico digital y con un nivel bajo de luz, un valor alto.

4.4. Encoders

Los encoders son sensores que convierten una posición lineal o angular en señal eléctrica o pulsos. Pueden determinar una posición de forma absoluta o simplemente informar que hubo un movimiento. El método de sensado y la resolución del ángulo de giro que detectan varía según el modelo.

4.4.1. Características

Los motores *MR-2FA* tienen encoders de cuadratura conectados al eje, previo a la caja reductora. Estos encoders son de tipo fotoeléctricos, estan dispuestos a 135° uno del otro y marcan 4 estados por cada vuelta del motor. Estan colocados así para poder conocer el sentido de giro midiendo la secuencia de estados de cada encoder. Adicionalmente el motor cuenta con un sensor de efecto Hall el cual nos permite detectar una revolución completa en el eje de salida de la caja reductora.

Nosotros como conocemos el sentido de giro del motor, pues lo determinamos con el puente H, necesitamos sólo uno de los dos encoders para conocer y controlar la velocidad a la que gira el rotor del motor.

A la tensión máxima los motores superan a las 320 cuentas por segundo, el mínimo y máximo recomendables son, para que podamos mantener un giro constante, 60 y 300 cuentas por segundo respectivamente. Estos cálculos son usando uno de los dos fototransistores del encoder.

Dependiendo el tamaño de las ruedas será la velocidad de final del robot. La relacioón de caja de $94:1$ del motor.

4.4.2. Circuito de control

Conectamos una resistencia pull-up a $5V$ para la salida de sensado de los fototransistores y del sensor de efecto Hall. También incluimos un switch doble inversor para elegir cuál de los dos encoders usar. El punto común del switch está conectado al pin de entrada de clock externo de uno de los Timers del microcontrolador.

La alimentación de los encoders es directa y permanecen encendidos en todo momento.

4.4.3. Rutinas de control

Como explicamos en la sección 3.1.3, en cada interrupción del actualizamos el histórico de cuentas del motor adicionando o restando el último valor del contador.

También comparamos contra las cuentas esperadas por intervalo de tiempo que fueron determinadas desde el controlador principal para poder determinar si debemos incrementar o disminuir la potencia del motor y por ende la velocidad de las ruedas.

4.5. Sensado de la bateria

El sensado del nivel de tensión en la batería lo hacemos mediante un divisor de tensión entre los polos de la batería. La salida del divisor es sensada de igual forma que los otros sensores, mediante el módulo de ADC del microcontrolador. En la figura 11 mostramos el diagrama del divisor de tensión.

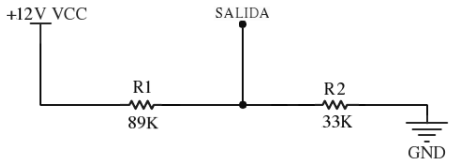

Figura 11: Divisor de tensión para el sensado de la batería.

En el cuadro 6 mostramos las posibles tensiones en la batería y la tensión de salida en el divisor. También incluimos el valor aproximado para el ADC con tensión de referencia a $5V$ que leería la salida del divisor. El rango de voltajes que analizamos es tienendo en cuenta la posibilidad de efectuar mediciones durante la carga de la batería y sabiendo que con una tensión menor a $5V$ la lógica del prototipo que construimos comenzaría a fallar.

Batería (V)	Salida (V)	Valor en el ADC
16	4.327	886
15	4.057	831
14	3.786	776
13	3.516	720
12	3.245	665
11	2.975	609
10	2.704	554
9	2.434	499
8	2.163	443
7	1.893	388
6	1.623	332
5	1.352	277

Cuadro 6: Tensión de la batería y la tensión de salida en el divisor.

La conexión es simple, los cables de entrada del divisor los conectamos a los polos de la batería y la salida al ADC. Luego sólo debemos realizar las lecturas

en el ADC y promediarlas para poder realizar los cálculos de la tensión en la batería.

4.6. Consumo del motor

El consumo de los motores de corriente contínua lo medimos leyendo el pin de sensado que se encuentra en el puente H que alimenta al motor. Lo que medimos con el módulo de *ADC* del microcontrolador es la tensión en este pin. Esta depende de la caída de tensión en la resistencia conectada a masa y de la corriente que circula por el motor. Conociendo el valor de la resistencia podemos calcular el consumo del motor. En el cuadro 7 comparamos el consumo en el motor, el voltaje sensado y la lectura en el ADC.

Tensión (V)	Consumo (A)	Valor en el ADC
0	0	0
0,09	0,19	18
0,18	0,38	36
0,27	0,57	55
0,36	0,76	73
0,45	0,95	92
0,54	1,14	110
0,63	1,34	129
0,72	1,53	147
0,81	1,72	165
0,90	1,91	184
0,99	2,10	202
1,08	2,29	221
1,17	2,48	239

Cuadro 7: Tabla comparativa para el consumo del motor.

4.6.1. Pulsador u otro dispositivo disparador

Ademas de los sensores que describimos, pensamos que nos podrían hacer falta pulsadores para controlar funciones simples por ejemplo saber cuando una parte de algún mecanismo llega a cierto punto o detectar finales de carrera de un servo o sin fin. También pueden ser otro tipo de sensores que generen un cambio de estado en el pin de sensado que se conecta al microcontrolador.

Agregamos la posibilidad de usarlos en las distintas placas como explicamos en detalle en las secciones 7.3 y 7.4.

4.6.2. Rutinas de control

La lectura en el estado de los pulsadores puede ser bajo demanda con solo leer el estado del pin en el que estan conectados o puede ser ante una interrupción por cambio de estado. Estas cuestiones las tuvimos en cuenta a la hora de diseñar las placas.

5. Controladores

Todas las funciones del robot las debíamos controlar mediante algún tipo de dispositivo. Decidimos utilizar una netbook y microcontroladores para esta tarea. Estas cuestiones son las que explicamos en este capítulo.

5.1. Netbook

Elegimos como controlador principal la netbook *EeePC 1005-HA* de la marca Asus[7]. Como características principales cuenta con un procesador Intel[8] Atom N280 1.66 GHz, 1GB DDR-2, un disco de 250GB, una pantalla de 10 pulgadas y una batería de 48Wh que nos da una autonomía de aproximadamente 8 horas. Cuenta con una cámara integrada de 0.3MP, placa de red inalámbrica y ethernet, placa de sonido y 3 puertos USB. Pesa aproximadamente 1.27Kg y sus dimensiones son 26,2 x 17,8 x 3,7 centímetros.

Usamos Ubuntu[9] como sistema operativo y programamos tanto la lógica de comportamientos como la captura y análisis de imágenes en C/C++.

5.2. Microcontrolador

Para el control de la velocidad de los motores, lectura de los encoders y los sensores de distancia usamos un microcontrolador. Nuestro diseño contempló la existancia de varios módulos con una o pocas funciones simples que se comunicaran entre ellos y con el controlador principal, en nuestro prototipo la netbook. La razón por la cual lo armamos así fue para simplificar cada placa controladora a nivel software y hardware.

Explicamos la comunicación entre los distintos controladores en la sección 6.

5.2.1. Características

El microcontrolador que elegimos para realizar las tareas de control, configucarión y comunicación a bajo nivel es el *PIC16F88* de Microchip[10]. Cuenta con una arquitectura de memoria del tipo Hardvard, con una memoria *FLASH* para 4096 instrucciones de programa, una memoria *RAM* de 368 bytes y una memoria *EEPROM* de 256 bytes. Tiene un set de instrucciones básicas reducido y todas con el mismo tiempo de ejecución. En este apartado nombramos algunos de los principales periféricos incluidos en el microcontrolador y la utilidad dentro del proyecto que encontramos para ellos. Utilizamos con un cristal externo de 20MHz como clock.

El microcontrolador tiene 2 puertos de 8 entradas y salidas cada uno de tipo TTL y CMOS. Como mostramos en la figura 12 cada pin se encuentra multiplexado con uno o más periféricos internos.

5.2.2. Módulos internos

Internamente el microcontrolador tiene una serie de periféricos que proveen de funciones extras y que utilizamos para lograr cumplir con las necesidades de

[7] http://www.asus.com/
[8] http://www.intel.com/
[9] http://www.ubuntu.com/
[10] http://www.microchip.com/

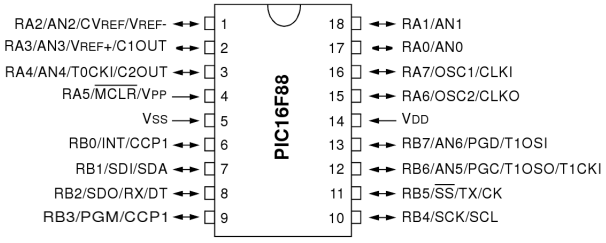

Figura 12: Diagrama del microcontrolador PIC16F88.

nuestro proyecto. En la figura 13 mostramos los distintos módulos internos del microcontrolador.

Cuenta con 3 timers, 2 de 8bits (*TIMER0* y *TIMER2*) y 1 de 16bits (*TIMER1*). Podemos configurarlos para que tomen al clock del microcontrolador o que tomen una fuente externa de clock. También podemos aplicarles demultiplicadores que generan un clock de menor frecuencia al que se usa como entrada. Pueden ser etapas previas o posteriores al timer y nos dan gran flexibilidad de uso.

El *TIMER0* lo utilizamos para hacer control del tiempo en nuestros códigos. Conectados con el clock principal y configurados para que generen una interrupción al hacer overflow, obtenemos una buena medida del paso del tiempo.

El *TIMER1* configurado como fuente externa a la salida del encoder, lo usamos como contador de pasos para medir la velocidad del motor. También lo usamos como medición del tiempo para hacer las lecturas de los sensores. Elegimos a este timer ya que al ser de 16bits posee un mayor rango de valores y por lo tanto, podíamos medir un mayor lapso de tiempo o cuentas del encoder en cada caso.

El *TIMER2* lo usamos en conjunto con el módulo de *PWM* para determinar el ancho del pulso que habilita al puente H que provee de energía a los motores.

El módulo conversor analógico digital nos dio la posibilidad de medir tensiones analógicas como por ejemplo las salidas de los sensores, tensión en la bateria o el consumo de los motores. Tiene 7 canales o pines distintos y podemos configurarlo para que genere un valor de 8 o 10bits. También podemos determinar si se debe usar el valor de *Vcc* y *GND* como referencia o podemos proveer de forma externa de los voltajes de referencia para generar un rango de voltajes diferente y aumentar o disminuir así la resolución del conversor.

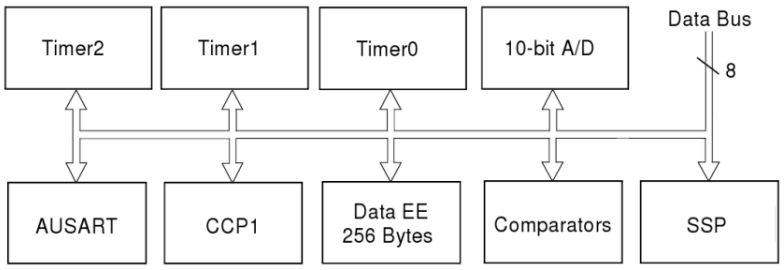

Figura 13: Módulos internos del microcontrolador PIC16F88.

Pines	Señal
1 y 2	MCLR
3 y 4	$5v$ VCC
5 y 6	GND
7 y 8	PGD (Data)
9 y 10	PGC (Clock)

Cuadro 8: Pines de programación en circuito con *ICD2*.

El módulo de *PWM* nos provee la posibilidad de generar pulsos contínuos de un ancho determinado. En nuestro proyecto lo utilizamos como habilitación del puente H que alimenta a los motores variando así la potencia y por lo tanto, la velocidad final de las ruedas. Se utiliza en conjunto con el *TIMER2* seteando el prescaler y postscaler para determinar el ancho del estado alto y del estado bajo de los pulsos.

Gracias al módulo de *AUSART* podemos realizar la comuinicación entre los distintos microcontroladores por hardware. Este periférico nos provee una comunicación sincrónica o asincrónica dependiendo de la configuración. Creamos la red de *Daisy Chain* sobre el protocolo de RS-232.

El microcontrolador dispone de otros periféricos como un comparador *CCP* y comunicación sincrónica *SPI*. Utilizamos los pines de uso general para realizar otras funciones como habilitación, conmutación o las señales de *PWM* por software para determinar la posición de los servo motores, por ejemplo.

5.2.3. Programación del firmware

El firmware de cada microcontrolador lo escribimos en C y usamos la IDE de programación *Microchip MPLAB*[11] con el compilador *CCS PCM V4.023*[12]. Usamos el programador *ICD2* de la empresa *Microchip* para descargar código compilado al microcontrolador. El cual nos dio la posibilidad de debuguear el código en más de una oportunidad. Mediante el header de programación en circuito que detallamos en el cuadro 8, establecemos la interface con el programador *ICD2* para cargar el firmware en el microcontrolador o para debuguearlo.

6. Comunicación

La comunicación interna entre los controladores de cada periférico y el controlador principal, donde concentramos la lógica de los comportamientos era vital. Para poder tomar las decisiones adecuadas la telemetría debía tener toda la información requerida, la frecuencia suficiente para que las reacciones sean lo más dinámicas y rápidas posibles y poseer la menor cantidad de errores para evitar retransmisiones.

En esta sección analizamos todo lo referente al medio de transmisión, protocolo y comandos que forman parte de la comunicación interna del proyecto.

[11] http://www.microchip.com/
[12] http://www.ccsinfo.com/

6.1. Conectividad entre módulos

Para establecer el canal de comunicaciones decidimos emplear una configuración basada en el método *Daisy-Chain*[13] entre los controladores. Creando un anillo donde cada nodo de la cadena se comunica con su vecino retransmitiendo cada paquete hacia adenlante hasta su destinatario como mostramos en la figura 14.

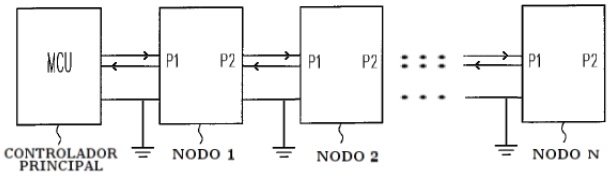

Figura 14: Diagrama general del método daisy chain

La configuración que elegimos para realizar la comunicación fue una velocidad de 115200 baudios, 8 bits, con 1 bit de parada, sin bit de paridad y sin control de flujo. Logramos una gran velocidad de respuesta a los comandos de esta forma.

En los cuadros 9 y 10 especificamos el conexionado entre las placas y contra el controlador principal.

Función	Conector RJ11	Conector RJ11	Función
Serial RX	2	2	Serial TX
Serial TX	3	3	Serial RX
No conectado	4	4	No conectado
GND	5	5	GND

Cuadro 9: Conexionado entre placas en modo Link

Función	Conector RJ11	Conector DB9	Función
Serial RX	2	3	Serial TX
Serial TX	3	2	Serial RX
No conectado	4	4	Shield
GND	5	5	GND

Cuadro 10: Conexionado entre placa y la PC

Como terminación de la cadena, debemos o colocar una ficha nula que interconecte *TX* con *RX* o cambiar el switch de configuración de *LINK* a *LAST*.

6.2. Protocolo de comunicación

El protocolo de comuncación está formado por paquetes que tienen un formato específico y representan un pedido de información o comando que debe ser ejecutado en el destino.

[13]Patente US20090316836A1

El paquete consta de un header común con datos que identifican al emisor y receptor del paquete, el comando a enviar y posibles datos extras que sean requeridos. Todos los paquetes tienen una respuesta obligatoria de confirmación de recepción. Cuando el paquete requiere una respuesta con datos, la confirmación va acompañada de la información requerida. En el cuadro 11 mostramos la estructura interna de un paquete típico.

LARGO	DESTINO	ORIGEN	COMANDO	DATO	CRC

Cuadro 11: Formato y header del paquete de datos

Tanto los paquetes de envío de datos como los de respuesta tienen el mismo formato y comparten el valor en el campo de comando.

El campo *LARGO* indica el tamaño en bytes del paquete que viene detras, consta de 1 byte. Necesario porque el campo *DATO* es de longitud variable, si la longitud del campo *DATO* es cero, es 0x04.

El campo *DESTINO* identifica al destinatario del paquete y consta de 1 byte. Los 4 bits más significativos indican el grupo y los 4 bits menos significativos, el número de ID de la placa de destino. Si el ID de placa es F, entonces el paquete es broadcast a todos los IDs del grupo indicado y si el valor es 0xFF, el paquete es broadcast a todas las placas de todos los grupos.

De igual forma *ORIGEN* determina el emisor del paquete para la respuesta y es de 1 byte. Los 4 bits más significativos indican el grupo y los 4 bits menos significativos, el número de ID de la placa de origen. Los valores permitidos son del 0 al E, ya que F indica broadcast y no es válida una respuesta broadcast.

El campo *COMANDO* informa al destinatario la tarea a realizar o determina un pedido de información que puede o no tener parámetros. Ocupa 1 byte.

El campo *DATO* contiene los parámetros o datos extras que puedan ser necesarios para el comando enviado. En el caso que el comando no los requiera, el campo debe ser nulo y el campo *LARGO* será 0x04.

El control de errores lo realizamos mediante un checksum calculado, haciendo un *XOR* con cada byte del contenido del paquete y colocandolo en el campo *CRC*. Cuando un paquete se encuentre con errores o esté mal formado, el destinarario debería pedir la retrasmisión. De igual forma, creimos necesaria la creación de un control adicional por medio de una lista de paquetes no confirmados mantenida por el controlador principal para evitar la pérdida de comandos.

6.2.1. Comandos comunes

Creamos comandos que son comunes a cualquier placa sea cual sea su grupo. La intención de esto fue poder tener un espacio de comunicación común para poder intercambiar comandos de inicialización, identificación de controladores, versionado o algúna otra necesidad genérica que se desarrolle en un futuro y deba ser incluida en el protocolo. El controlador principal debería ser quien envía este tipo de comandos, pero no creamos restricciones al respecto.

En el cuadro 12 listamos y explicamos brevemente los comandos comunes del protocolo.

Comando: *INIT*. Sincroniza el inicio de todas las placas en la cadena.	
0x01	Vacío
Respuesta	
0x81	Descripción de la placa en texto plano.
Comando: *RESET*. Pide el reset de la tarjeta.	
0x02	Vacío
Respuesta	
0x82	Descripción de la placa en texto plano.
Comando: *PING*. Envia un ping a la placa.	
0x03	Vacío
Respuesta	
0x83	Vacío
Comando: *ERROR*. Informa que ha habido un error.	
0x04	Código de error.
Respuesta	
0x00	Único comando sin respuesta directa. Paquete original.

Cuadro 12: Comandos comunes a todos los controladores.

6.2.2. Comandos específicos

Cada grupo de placas tiene comandos propios y específicos dependiendo de la función que deban desempeñar en el sistema. Existen grupos con comandos predefinidos cada uno se trata en las secciones como se detalla en el listado. Los comandos específicos para cada grupo deben ser desde 0x40 hasta el valor 0x7E.

El *MAIN CONTROLLER* no posee por ahora ningún comando escpecífico pero le reservamos el espacio en el protocolo. En los cuadros 13 y 14 enumeramos los comandos para el controlador *DC MOTOR*. El controlador *SERVO MOTOR* responde a los comandos de los cuadros y En el cuadro 17 listamos los comandos del controlador *DISTANCE SENSOR*. De igual forma, aunque no esten implementados reservamos y especificamos los comandos para *BATTERY CONTROLLER* y *TRASH BIN* en los cuadros 18 y 19 respectivamente.

6.2.3. Estadísticas

analisis de paquetes por segundo, bytes de datos vs bytes de header, retransmisiones, etc

7. Placas controladoras

Las funciones y requerimientos de nuestro proyecto tenían partes que eran comunes a otros trabajos en robótica o quizás de electrónica general. Pero tenían peculiaridades y aspectos que no pudimos satisfacer con placas prefabricadas. Por ejemplo, un protocolo de comunicación unificado entre todos los módulos era, de entrada, una barrera que acotaba las posibilidades. Y mantener distintos modos y protocolos dentro del proyecto generaba una carga de lógica que podiamos evitar.

SET DIRECTION. Seteo del sentido de giro del motor.	
0x40	Sentido de giro.
Respuesta	
0xC0	Vacío.

SET DC SPEED. Seteo de la velocidad del motor.	
0x01	Sentido de giro y velocidad en cuentas por segundos.
Respuesta	
0xC1	Vacío.

SET ENCODER. Seteo de las cuentas históricas del encoder.	
0x42	Valor para setear en el histórico del encoder.
Respuesta	
0xC2	Vacío.

GET ENCODER. Obtener las cuentas históricas del encoder.	
0x43	Vacío.
Respuesta	
0xC3	Valor histórico del encoder.

RESET ENCODER. Resetear las cuentas históricas a cero.	
0x44	Vacío.
Respuesta	
0xC4	Vacío.

SET ENCODER TO STOP. Seteo de cuantas cuentas para detenerse.	
0x45	Cuentas del encoder restantes para que el motor se detenga.
Respuesta	
0xC5	Vacío.

GET ENCODER TO STOP. Obtener la cuentas restantes hasta detenerse.	
0x46	Vacío.
Respuesta	
0xC6	Cuentas del encoder restantes para que el motor se detenga.

DONT STOP. Deshabilita el conteo de cuentas para frenar.	
0x47	Vacío.
Respuesta	
0xC7	Vacío.

MOTOR CONSUMPTION. Consulta sobre el consumo actual del motor.	
0x48	Vacío.
Respuesta	
0xC8	Consumo promedio del último segundo.

Cuadro 13: Comandos específicos al *DC MOTOR* parte A.

MOTOR STRESS ALARM. Alarma sobre un consumo extremo en el motor.	
0x49	Consumo ante el cuál sono la alarma.
Respuesta	
0xC9	Vacío.
MOTOR SHUT DOWN ALARM. Alarma, el motor ha sido apagado.	
0x4A	Consumo ante el que sono la alarma.
Respuesta	
0xCA	Vacío.
GET DC SPEED. Obtiene la velocidad en cuentas del encoder por segundo.	
0x4B	Vacío.
Respuesta	
0xCB	Sentido y velocidad de giro del motor.

Cuadro 14: Comandos específicos al *DC MOTOR* parte B.

Analizando las posibilidades y entendiendo que una de las principales razones de nuestro trabajo era generar las bases de conocimiento y el punto de partida a un proyecto más grande, decidimos producir nuestra propia versión de placas controladoras.

Aunque iban a tener un uso bien definido dentro de nuestro trabajo, también creímos necesario que tuvieran un nivel de generalidada suficiente como para poder utilizarlas en otros proyectos con mínimos cambios, adelantando así mucho trabajo a futuro.

La modularización fue un factor importante y muy presente durante todo el desarrollo. En este apartado explicamos el diseño, desarrollo, programación y especificaciones de las distintas placas que creamos durante nuestro proyecto.

7.1. Placa genérica

Durante el diseño nos surgió la necesidad de establecer un módulo común de comunicación y una base de prueba para los testeos con los distintos sensores y periféricos que utilizaríamos en el robot. Es por esto que luego de pruebas aisladas, creamos una placa para estas cuestiones. Nos ayudó y aceleró en gran medida el diseño de las placas definitivas y nos dió la posibilidad de diseñar futuras actualizaciones a nuestro proyecto.

7.1.1. Características principales

Las características principales eran proveer de una interfaz común entre todas las placas para la comunicación y exportar todos los pines y periféricos del microcontrolador que elegimos para placas con las conexiones a los sensores o motores. Todo esto nos servía realizar pruebas de concepto para la conexión con los sensores, valores de componente pasivos ideales, comportamiento e interacción entre placas, para optimizar el código del firmware o crear nuevas expansiones.

SET POSITION. Determina la posición del servo motor indicado.	
0x40	*ID* del servo y la posición del eje.
Respuesta	
0xC0	Vacío.

SET ALL POSITIONS. Setea las posiciones de cada uno de los servomotores.	
0x01	La posición para cada uno de los 5 servos.
Respuesta	
0xC1	Vacío.

GET POSITION. Obtiene la última posición del servomotor indicado.	
0x42	*ID* del servo del que se quiere la posición.
Respuesta	
0xC2	*ID* del servo y la posición del eje.

GET ALL POSITIONS. Obtiene todas las últimas posiciones de los servomotor.	
0x43	Vacío.
Respuesta	
0xC3	La posición para cada uno de los 5 servos.

SET SERVO SPEED. Setea la velocidad para el servomotor indicado.	
0x44	*ID* del servo y la velocidad a setear.
Respuesta	
0xC4	Vacío.

SET ALL SPEEDS. Setea la velocidad para cada servomotor.	
0x45	Velocidad a setear para cada uno de los servos.
Respuesta	
0xC5	Vacío.

GET SERVO SPEED. Obtiene la velocidad para el servomotor indicado.	
0x46	*ID* del servo.
Respuesta	
0xC6	*ID* y velocidad del servo.

GET ALL SPEEDS. Obtiene las velocidades de cada uno de los servomotores.	
0x47	Vacío.
Respuesta	
0xC7	Velocidad de cada uno de los servomotores.

FREE SERVO. Deja de aplicar fuerza sobre el servo indicado.	
0x48	*ID* del servo a liberar.
Respuesta	
0xC8	Vacío.

FREE ALL SERVOS. Deja de aplicar fuerza sobre cada uno de los servomotores.	
0x49	Vacío.
Respuesta	
0xC9	Vacío.

Cuadro 15: Comandos específicos al *SERVO MOTOR* parte A.

GET STATUS. Obtiene el estado de cada uno de los switches.	
0x4A	Vacío.
Respuesta	
0xCA	1 byte con el estado de cada switch.
ALARM ON STATE. Establece si se desea recibir alarmas por cambio de estado.	
0x4B	*ID* y tipo de cambio en el switch .
Respuesta	
0xCB	Vacío.
SWITCH ALARM. Alarma ante un cambio de estado programado.	
0x4C	*ID* y estado del switch que provocó el comando.
Respuesta	
0xCC	Vacío.

Cuadro 16: Comandos específicos al *SERVO MOTOR* parte B.

7.1.2. Módulo de comunicación

Como explicamos en la sección 6 para la comunicación nos basamos en el modelo de *daisy chain* sobre una capa de transporte de *RS232*.

Los conectores y configuración son comunes y utilizamos dos tipos de conectores para realizar la interconexión entre las placas. Como mostramos en la figura 15 usamos conectores *RJ11* para generar el lazo entre nodos de la cadena y el conector *DB9* para la conexión con la PC. En los cuadros 9 y 10 explicamos, respectivamente, el conexionado para cada caso.

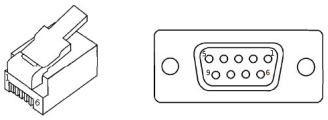

Figura 15: Conectores RJ11 (6P4C) y DB9.

Para determinar si la placa es un nodo más dentro de la cadena o es la terminación, colocamos una llave de dos posiciones que setea la configuración. En la posición *LINK* la placa actúa como nodo intermedio y en la posición *LAST* como terminación.

Es de vital importancia colocar en forma la correcta esta llave porque podemos dejar sin conexión al resto de las placas o mismo perder todos los paquetes transmitidos al no cerrar la cadena en el final de la misma.

7.1.3. Alimentación de la placa

La alimentación principal de la placa es 7 a 20 voltios, con la posibilidad de alimentarla directamente con 5 voltios por uno de los pines del conector. En la figura 16 mostramos la bornera y en el cuadro 20 el pinout.

La regulación interna de voltaje realiza por medio de un regulador 7805 corriente máxima de 1A.

ON DISTANCE SENSOR. Enciende el sensor de distancia indicado.	
0x40	ID del sensor a encender.
Respuesta	
0xC0	Vacío.
OFF DISTANCE SENSOR. Apaga el sensor de distancia indicado..	
0x01	ID del sensor a apagar.
Respuesta	
0xC1	Vacío.
SET DISTANCE SENSORS MASK. Habilita o no los sensores para lecturas.	
0x42	1 byte con cada ID del sensor a habilitar o deshabilitar.
Respuesta	
0xC2	Vacío.
GET DISTANCE SENSORS MASK. Obtiene la máscara de lectura.	
0x43	Vacío.
Respuesta	
0xC3	1 byte con cada ID del sensor.
GET VALUE. Obtiene el valor promedio de la entrada de los sensores indicados.	
0x44	1 byte con cada ID del sensor.
Respuesta	
0xC4	1 byte con cada ID y los valores de lectura promedio de cada sensor.
GET ONE VALUE. Obtiene el valor de la entrada del sensor indicado.	
0x45	1 byte con cada ID del sensor.
Respuesta	
0xC5	1 byte con cada ID y los valores de lectura de cada sensor.
ALARM ON STATE. Setea el tipo de cambio de estado del switch para la alarma.	
0x46	Tipo de cambio de estado.
Respuesta	
0xC6	Vacío.
SWITCH ALARM. Alarma informando que fue satisfecha la condición.	
0x47	Tipo de cambio y estado actual del switch.
Respuesta	
0xC7	Vacío.

Cuadro 17: Comandos específicos al *DISTANCE SENSOR*.

ENABLE. Habilita la alimentación del robot mediante la batería.	
0x40	Vacío.
Respuesta	
0xC0	Vacío.
DISABLE. Deshabilita la alimentación del robot mediante la batería.	
0x01	Vacío.
Respuesta	
0xC1	Vacío.
GET BATTERY VALUE. Obtiene el valor de la entrada de la batería.	
0x42	Vacío.
Respuesta	
0xC2	Lectura de la tensión en la batería.
BATTERY FULL ALARM. Mensaje informando que se completado la carga.	
0x43	Vacío.
Respuesta	
0xC3	Vacío.
SET BATTERY EMPTY VALUE. Valor de la batería crítico.	
0x44	Lectura de los voltios de la batería.
Respuesta	
0xC4	Vacío.
BATTERY EMPTY ALARM. El voltaje llegó a un valor crítico.	
0x45	Lectura de los voltios de la batería.
Respuesta	
0xC5	Vacío.
SET FULL BATTERY VALUE. Establece el valor para la carga completa.	
0x46	Lectura de la tensión en la batería.
Respuesta	
0xC6	Vacío.

Cuadro 18: Comandos específicos al *BATTERY CONTROLLER*.

GET TRASH BIN VALUE. Consulta que tan lleno está el cesto de basura.	
0x40	Vacío.
Respuesta	
0xC0	Valor que representa que tan lleno está el cesto interno de basura.
BIN FULL ALARM. El cesto de basura se ha completado y debe ser descargado.	
0x01	Vacío.
Respuesta	
0xC1	Vacío.
SET FULL BIN VALUE. Setea el valor para determinar que el cesto esta lleno.	
0x42	Valor que especifica que lecura se debe tomar como cesto lleno.
Respuesta	
0xC2	Vacío.

Cuadro 19: Comandos específicos al *TRASH BIN*.

Figura 16: Bornera de alimentación.

Pin	Voltaje
1	GND
2	5v
3	7v a 12v

Cuadro 20: Alimentación de la lógica

7.1.4. Configuración

La fila de pines *P2* exporta todos los pines con funciones dentro del microcontrolador, para realizar conexiones con periféricos de prueba o nuevas extensiones. Los headers *P3* y *P4* son jumpers que vinculan los pines *RA1* y *RA4* del microcontrolador los leds 1 y 2 respectivamente.

El header de programación *P1* lo utilizamos para conectar la placa con el programador y debuguear de código *ICD2* como explicamos en en apartado 5.2.3. El switch *S2* lo usamos para asociar los pines del microcontrolador con los canales de clock y data del header de programación (modo *ICD2*) o con los pines del header *P2*.

7.1.5. Esquemático

En la figura 17 mostramos el esquemático del microcontrolador y el conexionado con los headers, el módulo de comunicación y conectores para conformar la cadena *Daisy chain* los detallamos en la figura 18. En la figura 19 mostramos el diagrama de la fuente de alimentación y bornera.

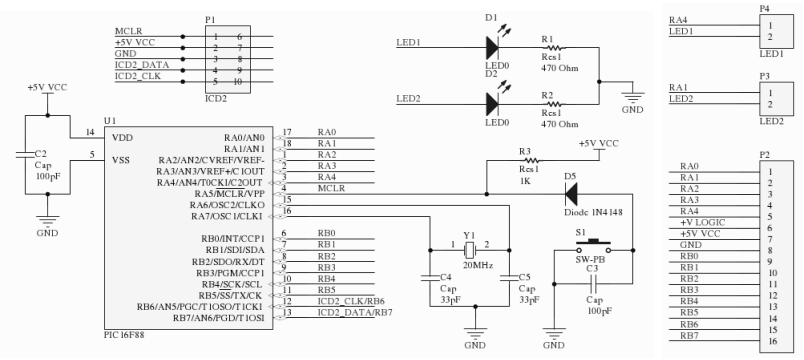

Figura 17: Microcontrolador y headers

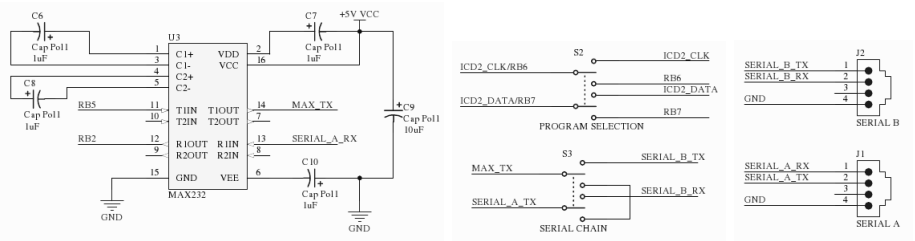

Figura 18: Comunicación, switch de modo y conectores de entrada y salida

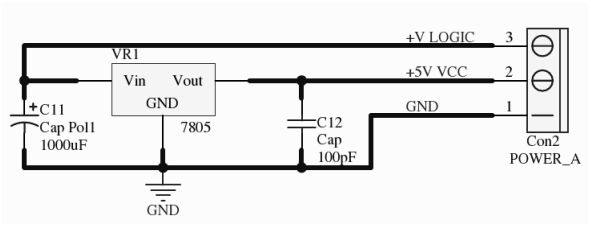

Figura 19: Fuente de alimentación

7.1.6. Circuito

En la figura 20 mostramos la máscara de componentes de la placa y en la figura 21 ambas capas de la placa.

7.1.7. Código básico

Según el uso que le fuimos dando a las placas genéricas fuimos cambiando el código del firmware que le íbamos cargando, pero siempre manteníamos partes básicas que nos ayudaban.

Principalmente la configuración de flags de configuración base del microcontrolador, desde la velocidad del clock para los cálculos de tiempos hasta determinar que pines eran mapeados a las funciones de RX y TX para la comunicación serial.

Para hacer pruebas sobre la comunicación RS232 o del protocolo, armamos maquetas para las funciones que atienden a los comandos específicos y funciones de control del protocolo comunes para todas las placas. También generamos un modelo general del programa, con un loop principal y llamadas incializadoras. De esta sólo nos concentrábamos en la prueba que estábamos realizando.

7.1.8. Posibles extensiones

Son varias las extensiones que se podrían realizar a estas placas pero teniendo en cuenta que debían ser lo más genéricas posibles como para probar nuevas funcionalidades o componentes, creemos que aportan mucho valor en la fase de testeo. Aunque usar componentes de montaje superficial nos permitirían tener una placa mas compacta.

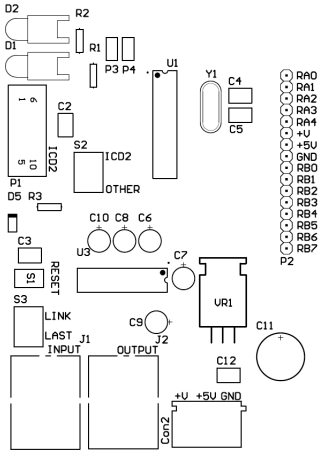

Figura 20: Máscara de componentes de la placa genérica.

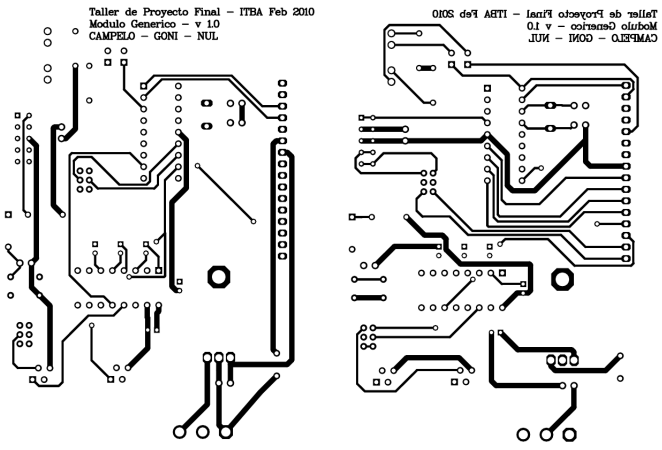

Figura 21: Capas superior e inferior de la placa genérica.

7.2. Placa controladora de motores DC

Nuestro proyecto implicaba, entre otras cosas, el diseño y construcción de un robot movil el cual devía poder trasladarse por el terreno como primer requerimiento. Y debido a que el desplazamiento iba a ser mediante ruedas, el control de los motores que las impulran era crítico.

Creamos una placa especializada para esta tarea sobredimensionando los requerimientos para no tener acotadas las opciones a futuro sobre qué modelo de motores usaríamos.

En este apartado explicamos los aspectos que tuvimos en cuenta para diseñar y crear la placa controladora de motores de corriente continua para el robot.

7.2.1. Características principales

La función principal es entre otros, mantener una velocidad estable en el motor de contínua, pero también poder sensar la cantidad y sentido del movimiento que realizó la rueda, informar el consumo y poder determinar el desplazamiento a realizar. Este último requerimiento era de vital importancia si queríamos tener precisión en el movimiento que realizaría la rueda ya que el control de alta resolución hubiera sido imposible hacerlo desde el controlador principal.

Todos los comandos desde y hacia el controlador principal viajan por la red de comunicación como explicamos en la sección 6 colocándose como un eslabón más en la cadena. Usamos como base la placa genérica que explicamos en la subsección 7.1 y agregamos el circuito necesario para el control del motor como explicamos en el punto 7.2.6.

7.2.2. Comunicación

Al utilizar la placa genérica como base mantuvimos la sección de comunicación de la misma intecta, pues en principio, era un módulo probado y funcional, además manteníamos la misma estructura física que facilitaba el conexionado para futuros usuarios.

Como explicamos en la sección 6.2.2 esta placa responde a comandos específicos listados en los cuadros 13 y 14. La principal información disponible mediante estos comandos es la velocidad de la rueda en cuentas del encoder por segundo, el sentido de giro y el histórico de cuentas realizadas. Podemos enviar comandos para setear estos valores y otros, como la cantidad de cuentas del encoder y a que velocidad debe girar la rueda debe hacerlo.

También podemos leer la cantidad de cuentas restantes antes de frenar y el consumo actual del motor para hacer cálculos sobre el tiempo restante de batería.

7.2.3. Alimentación de la placa

Como explicamos en la subsección 7.1, la alimentación principal de la lógica es de 7 a 20 voltios, con la posibilidad de alimentarla directamente con 5 voltios por uno de los pines del conector. La alimentación del motor es por una bornera de 2 pines como listamos en el cuadro 21, aislada completamente para disminuir así el ruido generado por el funcionamiente del motor sobre de la lógica.

Como la alimentación del motor se hace mediante el *puente H*, no puede superar los $46V$ que es el máximo que puede operar el *L298*.

Pin	Voltaje
1	GND
2	12v (depende del motor)

Cuadro 21: Pines de alimentación del motor.

Pin	Señal	Pin	Señal
1	MOTOR_B	2	MOTOR_A
3	IDX	4	-
5	ENCODER_B	6	GND
7	ENCODER_A	8	GND
9	$5v$ VCC	10	GND

Cuadro 22: Pines del header de comunicación con el motor.

Cabe aclarar también que al estar ambos circuitos completamente aislados, se necesita unificar ambas masas para establecer un punto de referencia común para poder ejercer el contro del *puente H* y lectura del consumo, de forma correcta. Recomendamos hacer esto lo más cerca posible de la batería para disminuir al máximo el ruido que pueda generarse debido a los motores.

7.2.4. Configuración

La configuración de la placa es relativamente sencilla. Antes que nada debemos determinar el rol de la placa dentro de la cadena de comunicación, esto lo hacemos mediante el switch *S3*. Podemos colocarlo en modo *LINK* para que sea un eslabón más o en modo *LAST* para que sea la terminación de la cadena.

El header del motor comunica la placa con el motor de contínua, en el cuadro 22 detallamos la posición de los pines.

Las señales *MOTOR_B* y *MOTOR_A* son para la alimentación del motor, dependiendo del sentido de circulación la corriente entre estos pines será el sentido de giro del motor. Esto se controla desde el puente H *L298*.

Los sensores dentro del motor están reciben tensión mediante los pines $5v$ *VCC* y *GND*. La señal *IDX* genera pulsos según las vueltas en el eje de salida de la caja de reducción del motor. En cambio las señales *Encoder_A* o *Encoder_B* generan pulsos en base a las vueltas del motor antes de entrar en la caja.

El switch *S2* lo utilizamos para asociar los pines del microcontrolador con el canal de datos del header de programación, que explicamos en la sección 5.2.3, modo *ICD2* o con la señal de lectura del encoder.

La señal del encoder proviene a su vez del switch *S4*, el cual nos da la posibilidad de elegir entre alguna de las dos señales *Encoder_A* o *Encoder_B*.

Los pines *P2* y *P3* los usamos como referencia del conversor analógico digital respectivamente, un puente a *GND* y *VCC*, o al divisor de tensión formado por las resistencias *R5* y *R6*.

7.2.5. Esquemático

En la figura 22 mostramos el esquemático del microcontrolador y el conexionado el header de programación. En la figura 23 detallamos el módulo de

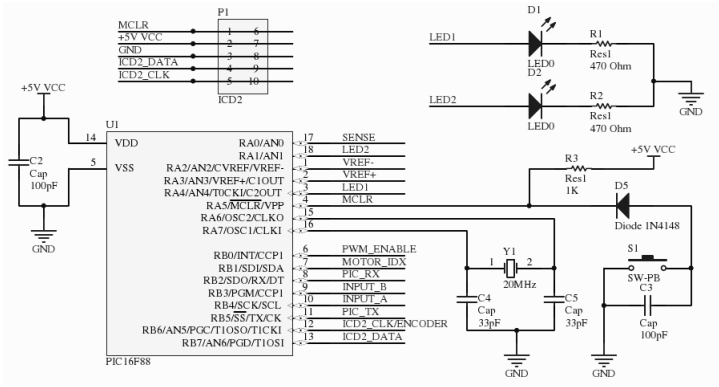

Figura 22: Microcontrolador y header de programación.

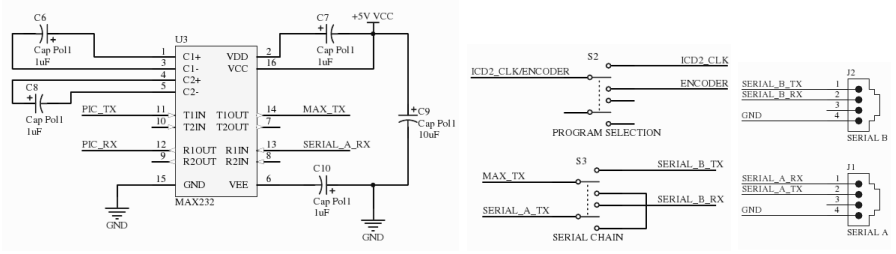

Figura 23: Comunicación, switch de modo y conectores de entrada y salida.

comunicación y el conexionado con los conectores entre placas. En la figura 24 el esquemático del driver y el header del motor. En la figura 25 mostramos los pines del divisor de tensión para el conversor analógico digital y en la figura 26 la fuente de alimentación y borneras.

7.2.6. Circuito

En la figura 27 se muestra la máscara de componentes de la placa. En la figura 28 se muestran ambas capas de la placa.

7.2.7. Código básico

El código mínimo para ser parte de la cadena de comunicación es similar al de todas las placas que armamos. La personalización del codigo está dentro de la función que interpreta el comando recibido y realiza las tareas internas necesarias para llevar a cabo dicha acción y depende íntegramente del protocolo.

Para el sensado de la velocidad debemos determinar cuales son los pines de lectura de la señal del encoder y debemos generar una base de tiempo que nos permita estableces cuantas cuentas del encoder hay por intervalo de tiempo. Con dicha información podemos aumentar o disminuir el ancho del pulso que controla al puente H *L298* y, en consecuencia, controlar la velocidad del motor de corriente contínua.

Para mantener un histórico de cuentas de encoder realidas nosotros usamos el conocimiento del sentido de giro del motor para sumar o restar el valor calculado

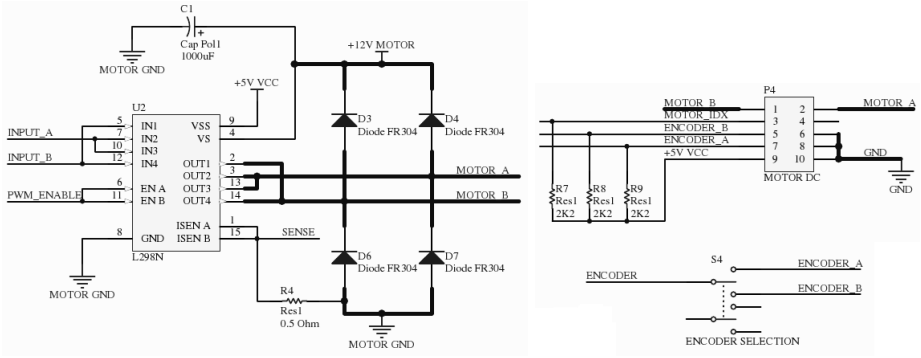

Figura 24: Driver y header de conexión con el motor.

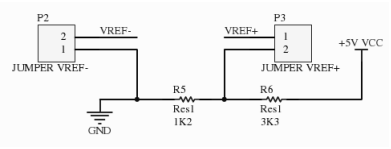

Figura 25: Divisor de tensión para el voltaje de referencia.

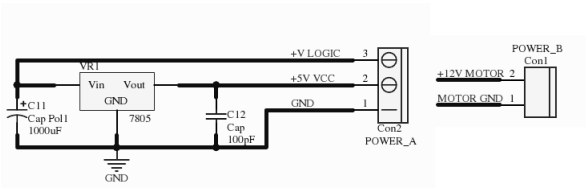

Figura 26: Fuente de alimentación de la lógica y motor.

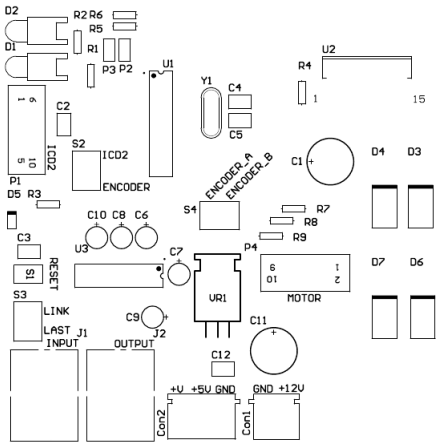

Figura 27: Máscara de componentes de la placa controladora de motores DC.

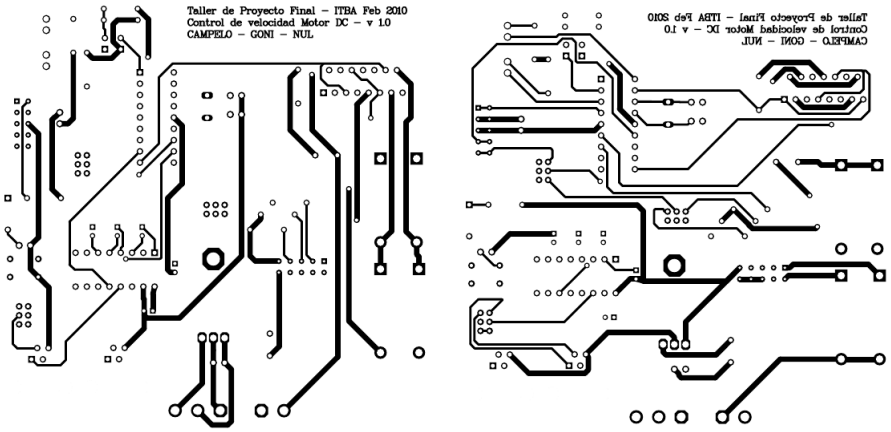

Figura 28: Capas superior e inferior de la placa controladora de motores DC.

por cada intervalo de tiempo en una variable interna del microcontrolador.

7.2.8. Posibles extensiones

Un punto recurrente de discusión fue el uso de una placa por cada motor de corriente contínua. Sabíamos que era posible realizar todas las funciones necesarias para el control de dos motores en una única placa, pero esto nos traía una mayor complejidad tanto a nivel físico como a nivel lógico de control, y un tiempo de implementación superior. Es por estas razones que decidimos esta configuración y dejarlo como futura extensión para la cual usaríamos los conocimientos y experiencias adquiridas en el desarrollo de la actual.

Otra posible extensión que es recurrente en todas las placas es el uso de componentes de montaje superficial lo que disminuiría el tamao de la misma en gran medida pero a su vez implicaba cierta complejidad que preferimos evitar en el desarrollo del prototipo.

7.3. Placas de sensado

Para sensar el entorno utilizamos distintos tipos de sensores que debíamos leer y controlar según fuera requerido desde el controlador principal. Diseñamos una placa a la que podemos conectarle cualquiera de los sensores que usamos en nuestro proyecto, ya sea un sensor de distancia por ultrasonido, telémetros o sensores reflectivos de piso. Tambíen utilizamos esta placa para realizar las lecturas de la tensión de la batería. En este apartado detallamos cada una de las características que tuvimos en cuenta durante el desarrollo de las placas de sensado.

7.3.1. Características principales

El uso de diferentes tipos de sensores implicaba un método distinto de lectura según qué sensor estaba conectado. Los sensores de distancia por ultrasonido que usamos generan un pulso de ancho variable según la distancia al objetivo, de

forma similar, los telémetros setean un voltaje en la salida en base a la distancia con el objeto. Los sensores reflectivos de piso que usamos miden el nivel de luz infrarroja que refleja la superficie que tienen debajo de ellos y en base al valor sensado inferimos su color.

Los sensores que devuelven el valor sensado mediante un voltaje utilizamos el conversos analógico digital para tomar las muestras necesarias. Salvo que se indique lo contrario, tomamos 5 muestras y luego las promediamos para evitar un muestreo erróneo.

En el caso del sensor de ultrasonido usamos el módulo de timer para calcular el ancho del pulso generado por el sensor. También usamos interrupciones para captar los flancos ascendentes y descendentes para determinar el comienzo y fin del pulso respectivamente.

Armamos la placa para poder conectar 1 sensor de distancia por ultrasonido o un switch binario sobre el mismo puerto de conexión. De manera conjunta podemos conectar a la misma placa, hasta 5 sensores por voltaje, telémetros, fototransistores (sensores de piso) o por ejemplo el divisor de tensión que mide la carga en la batería.

Generamos la posibilidad de habilitar y deshabilitar la alimentación de estos sensores para poder aumentar así la autonomía del robot. Mediante el uso de un transistor *BC327* por el que puede circular una corriente de hasta $500mA$ para alimentar al sensor que sea necesario.

7.3.2. Módulo de comunicación

De igual forma que las otras placas en el proyecto, mantenemos la misma disposición física de la comunicación, desde el circuito base hasta la ubicación de los conectores para mayor facilidad a la hora de utilizarlas. También respetamos el protocolo de comunicación definiendo nuevos comandos específicos para la placa y retransmitiendo los paquetes con otro destinatario.

7.3.3. Alimentación de la placa

De igual forma que las otras placas la alimentación principal de la lógica es de 7 a 20 voltios, con la posibilidad de alimentarla directamente con 5 voltios por uno de los pines del conector como explicamos en la subsección 7.1.

7.3.4. Configuración

Para la configuración física de la placa necesitamos definir primero el rol de la placa dentro de la cadena de comunicación con la llave *S3*. Como en todas las placas que diseñamos sólo tenemos que elegir el modo *LAST* o *LINK* para determinar si es la terminación o un eslabón más en la cadena.

Debemos saber que no podemos conectar más de 5 sensores de los cuales debamos medir una tensión para obtener el valor y sólo un sensor de distancia por ultrasonido o switch en el puerto para medir estados lógicos y tiempo del pulso.

Con la llave *S2* cambiamos al modo programación vinculando el microcontrolador con el header de *ICD2* o con la habilitación de los puertos de los sensores *3* y *4*. Si esta llave esta en la posición incorrecta nunca se exitará el transistor y por ende no habrá alimentación en el sensor.

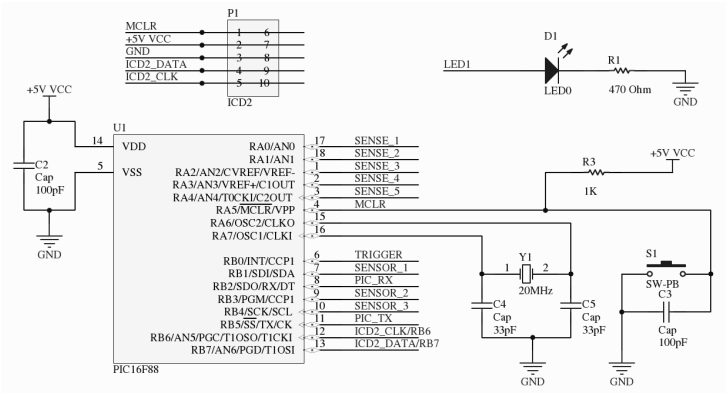

Figura 29: Microcontrolador y header de programación.

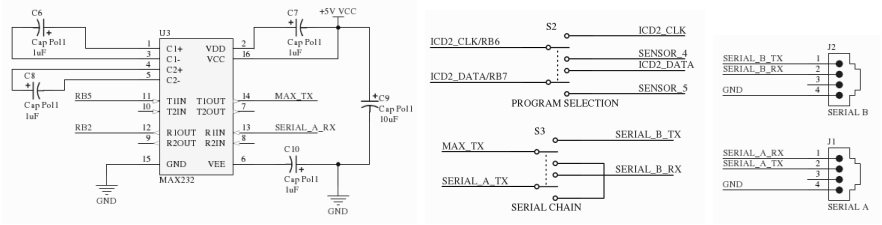

Figura 30: Comunicación, llaves de modo y conectores de entrada y salida.

En el caso que necesitemos una resistencia *pull-up* conectada con la salida del sensor podemos colocar un jumper en el correspondiente header de pull-up del puerto de sensor.

7.3.5. Esquemático

En la figura 29 mostramos el esquemático del microcontrolador y el conexionado el header de programación. En la figura 30 detallamos el módulo de comunicación y el conexionado con los conectores entre placas.

En la figura 31 mostramos los puertos de conexión para cada uno de los sensores y en la figura 32 la fuente de alimentación y bornera.

7.3.6. Circuito

En la figura 33 se muestra la máscara de componentes de la placa. En la figura 34 se muestran ambas capas de la placa.

7.3.7. Código básico

Al compartir el módulo de comunicación entre las placas y para que nuestra placa sea parte de la cadena y podamos interpretar los paquetes que viajan por ella, debemos incluir el código común de comunicación e implementar la función que analiza y ejecuta los paquetes que son dirigidos directamente a nuestra placa, grupo o a nivel broadcast.

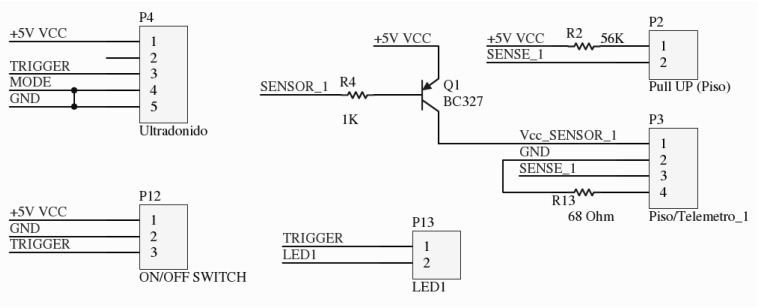

Figura 31: Puertos de conexión para los sensores y header de *pull-up*.

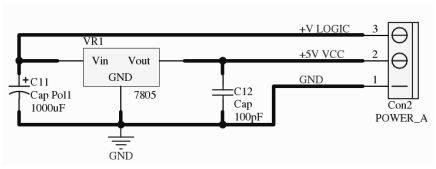

Figura 32: Fuente de alimentación de la lógica.

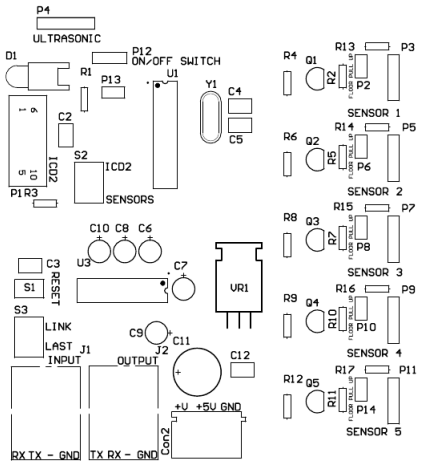

Figura 33: Máscara de componentes de la placa controladora de sensores.

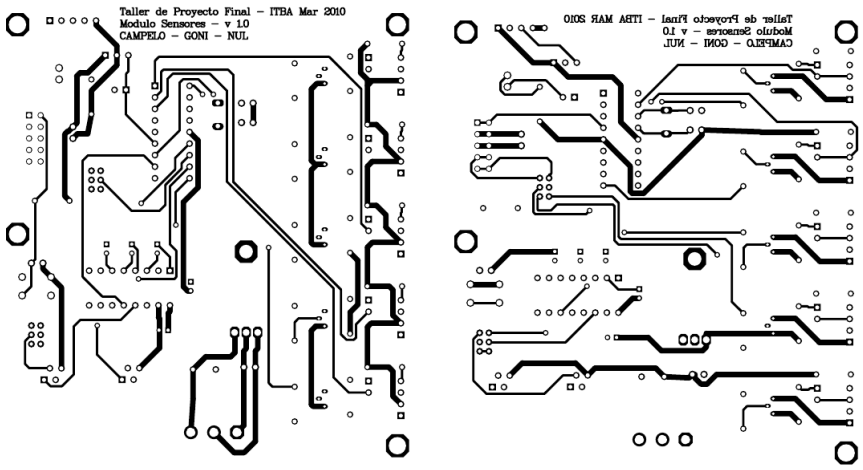

Figura 34: Capas superior e inferior de la placa controladora de sensores.

Dependiendo de la selección de los sensores a conectar será la configuración interna del microcontrolador. Al principio del código colocamos una serie de scripts que nos ayudan a configurar los tiempos a los que se deben tomar las muestras de los sensores y los flags del microcontrolador.

Los pines que exitan a la base de los transistores usan una lógica inversa, es decir, un estado bajo habilita la alimentación del sensor y un estado alto, lo deshabilita.

7.3.8. Posibles extensiones

Dentro de las posibles extensiones que creemos que serían útiles a futuro podemos decir que la más significativa sería usar potenciómetros en vez de resistencias fijas para usar como *pull-up* y para variar la alimentación de los sensores de piso por ejemplo.

En una implementación futura también agregaríamos el uso de componentes de montaje superficial y aumentaríamos la cantidad de puertos para sensores disponibles. De igual forma, incorporar otro tipo de sensores nos parece una interesante mejora.

7.4. Placa controladora de servo motores

Para los mecanismos que implicaran algún tipo de movimiento y control de la posición elegimos usar de servo motores, ya sea para el mecanismo de recolección de basura, inclinar el robor, reclinar el cesto interno de basura o realizar algún movimiento de paneo o foco con la cámara.

Diseñamos esta placa teniendo en cuenta que usaríamos varios servos en el robot y que también necesitaríamos tener la posibilidad de recibir señales del estilo *fin de carrera* o algún estímulo generado por pulsadores que modificaran o no el accionar de los motores.

Esta placa no la construimos para este prototipo debido a que no contábamos con la necesidad. Al no implementar ningún método físico de recolección, no

teníamos mecanismos que la usaran, aunque sí entendimos que era algo que debíamos hacer para tener el conocimiento y sentar una base para futuras implementaciones o proyectos similares.

7.4.1. Características principales

Diseñamos la placa pensando en que tendríamos que controlar 5 servo motores. Debido a que pensamos usar servo motores de tipo digital necesitabamos varios módulos de PWM para realizar esta tarea. Pensamos en el uso de otros microcontroladores que pudieran proveernos de esta facilidad, pero finalmente decidimos implementar esta funcionalidad por software.

A los puertos que no estan destinados al control de los servo motores les podemos dar varios usos. En un principio los pensamos como generadores interrupciones ante un cambio en el estado y en el puerto *P9* contamos con la posibilidad de determinar el flanco ante el cual se generará la interrupción. En el puerto *P8* podemos colocar algún tipo de sensado que haga uso del conversor analógico digital del microcontrolador.

De igual forma podríamos extender la rutina que genera el PWM por software, a todos los puertos y controlar hasta 11 servos con una única placa.

7.4.2. Módulo de comunicación

Como utilizamos como base la otras placas y mantuvimos la sección de comunicación, para ser parte de la cadena sólo debemos incluir el código común e implementar las acciones necesarias dentro de la función que interpreta los comandos a los que somos destinatarios.

Como explicamos en la sección 6.2.2 creamos ciertos comandos o paquetes específicos para el control de los servos. De igual forma contemplamos el uso de pulsadores en la misma.

Un mayor cambio a nivel funcional implicaría que hicieramos otros cambios a nivel protocolo, el cual lo pensamos para ser expandido creando nuevos comandos o modificando los existentes.

7.4.3. Alimentación de la placa

Al igual que las otras placas utilizamos entre 7 y 20 voltios para alimentar la placa, con la posibilidad de entregar $5v$ en forma directa usando el pin indicado.

Hicimos una modificación importante para la alimentación en esta placa que nos permitiría entregar una mayor cantidad de corriente que los $500mA$ que soporta el integrado *7805* usando transistores de potencia como los *TIP42*. Este cambio era importante porque los servo motores que íbamos a usar se alimentaban directamente a $5v$ y el consumo en conjunto superaba la cantidad de corriente que el regulador de tensión entregaba.

7.4.4. Configuración

La configuración física necesaria en la placa sería mínima. Mediante la llave inversora *S2* tendríamos la posibilidad de conectar el microcontrolador con el header de programación o puertos *P12* y *P13*.

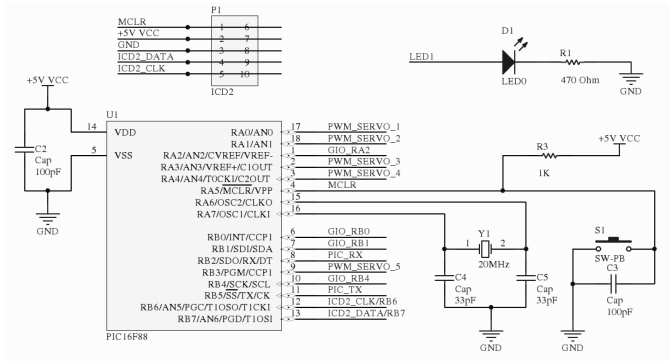

Figura 35: Microcontrolador y header de programación.

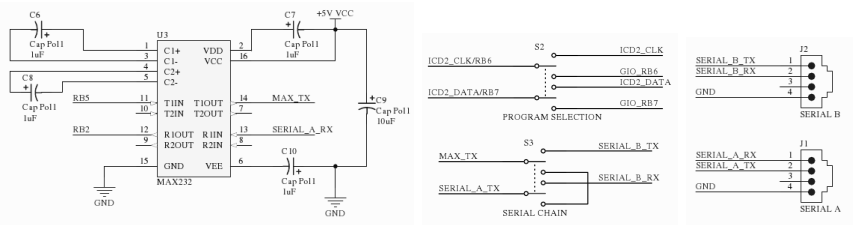

Figura 36: Comunicación, llaves de modo y conectores de entrada y salida.

Usando la llave *S3* estableceríamos la posición de la placa dentro de la cadena, ya sea como la terminación o un eslabón más. Todas las demás configuraciones serían a nivel software.

7.4.5. Esquemático

En la figura 35 mostramos el esquemático del microcontrolador y el conexionado el header de programación. En la figura 36 detallamos el módulo de comunicación y el conexionado con los conectores entre placas.

En la figura 37 mostramos los puertos de conexión para los servo motores y de uso general para otras conexiones. En la figura 38 la fuente de alimentación y bornera.

7.4.6. Circuito

circuito de la placa En la figura 39 se muestra la máscara de componentes de la placa. En la figura 40 se muestran ambas capas de la placa.

7.4.7. Código básico

Como explicamos en la sección 3.2.2 creamos una rutina de control que usando el timer del microcontrolador como base de tiempo, genera pulsos del ancho necesario en cada puerto para establecer la posición necesaria de cada servo motor.

Para capturar cambios de estado en los puertos de uso generar sólo deberíamos configurar las interrupciones del microcontrolador para que se generen

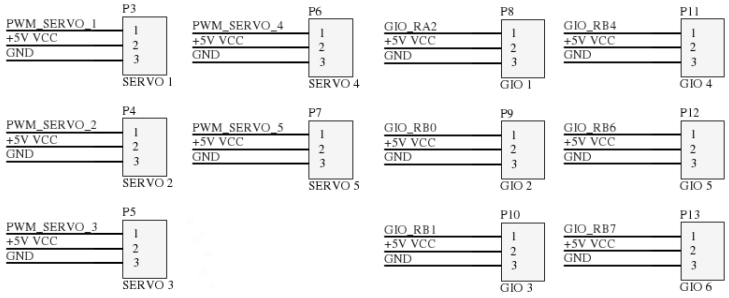

Figura 37: Puertos de conexión para los servo motores y de uso general.

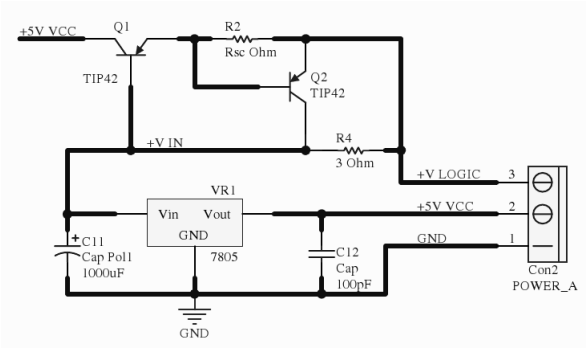

Figura 38: Fuente de alimentación extendida para la lógica y servo motores.

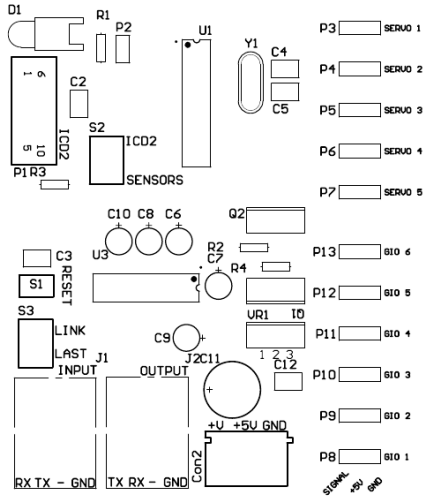

Figura 39: Máscara de componentes de la placa controladora de servo motores.

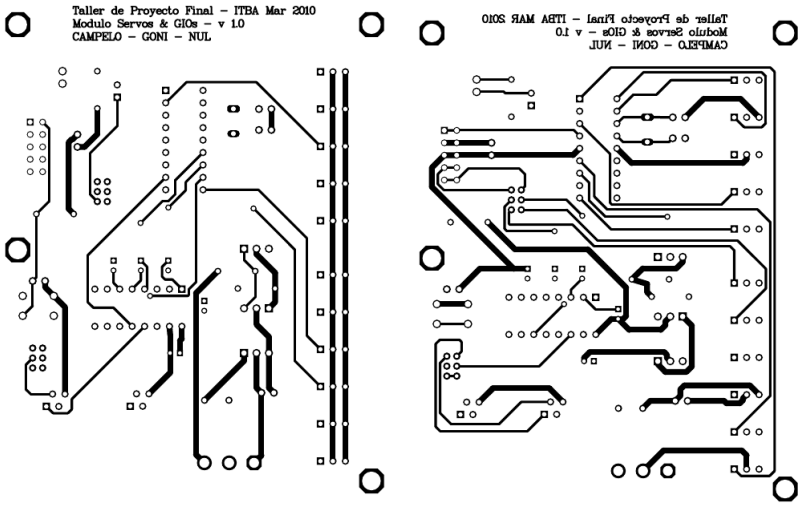

Figura 40: Capas superior e inferior de la placa controladora de servo motores.

ante cualquier cambio y verificar en la rutina de interrupción, que puerto la generó.

De querer utilizar los puertos para funciones no previstas en nuestro diseño deberíamos agregar las funciones de control en el ciclo principal de nuestro código.

7.4.8. Posibles extensiones

Posiblemente haya necesidades que no contemplamos en es diseño de esta placa aunque por no haberla construido no tenemos la experiencia para recapacitar qué cosas hubieramos hecho de alguna otra forma.

8. Armado del prototipo

Una vez diseñadas, armadas y probadas cada una de las placas controladoras tanto en forma separada como en conjunto, procedimos al armado del robot prototipo.

fisico: tamaño debía tener? forma? xq? que materiales usar?

bateria: que autonomía necesitabamos? tamaño y peso de la batería? como cargarla?

Sensores: Cuantos telemetros? cuantos sensores de piso? ultrasonido? donde ponerlos? A qué distancia entre ellos ponerlos?

Motores: donde ponerlos?

Ruedas: Tamaño? tipo de rueda? problemas con los ejes

Placas: Donde ponerlas? como sostenerlas?

Netbook: Donde ponerla? autonomia? como cargarla?

8.1. Diseño

8.2. Características

con las ruedas de 10cm y teniendo en cuenta que el motor gira a unas 300 cuentas/segundo (usando un solo sensor en el encoder) llegamos a 50 cm/s de velocidad

8.3. Desarme

A. protocolo de comunicacion

B. Código fuente

En este apartado mostramos el código fuente usado para cada una de las placas controladoras del proyecto. Pequeños cambios existen en *ID* de grupo y placa fueron necesarios para poder que sean únicas dentro de la cadena de comunicación.

B.1. Placa genérica

A la placa genérica la pensamos para testeo y como punto de partida de nuevas implementaciones, agregamos el código que usamos como base para la programación de las otras placas.

```
#define CARD_GROUP      MOTOR_DC        // Ver protocol.h
#define CARD_ID         0               // Valor entre 0 y E

// Descripcion de la placa
#define DESC            "PLACA GENERICA - 1.0" // Maximo DATA_SIZE bytes

/* Modulo Generico - main.c
 * PIC16F88 - MAX232 - GENERICO
 *
 *                                PIC16F88
 *                .-----------------------------------.
 *               -|RA2/AN2/CVREF/VREF           RA1/AN1|- LED
 *               -|RA3/AN3/VREF+/C1OUT          RA0/AN0|-
 *           LED -|RA4/AN4/T0CKI/C2OUT    RA7/OSC1/CLKI|- XT CLOCK pin1, 27pF to GND
 * RST/ICD2:MCLR -|RA5/MCLR/VPP           RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *           GND -|VSS                           VDD|- +5v
 *               -|RB0/INT/CCP1       RB7/AN6/PGD/T1OSI|- ICD2:PGD/
 *               -|RB1/SDI/SDA  RB6/AN5/PGC/T1OSO/T1CKI|- ICD2:PGC/
 *  MAX232:R1OUT -|RB2/SDO/RX/DT           RB5/SS/TX/CK|- MAX232:T1IN
 *               -|RB3/PGM/CCP1             RB4/SCK/SCL|-
 *                '-----------------------------------'
 *
 */

#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// Led
#bit led1=porta.1
#bit led2=porta.4

// MAX232
#bit tx=portb.5
#bit rx=portb.2

#include <../../protocolo/src/protocol.c>
/*
** Variables definidas en protocol.c
```

```
short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE];   // Buffer de recepcion de comandos
int buffer_write;               // Indice de escritura
int buffer_read;                // Indice de lectura
int data_length;                // Largo de los datos en el buffer

struct command_t command;       // Comando parseado
struct command_t response;      // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecula el comando

***/

void init()
{
        // Inicializa puertos
        set_tris_a(0b11100101);
        set_tris_b(0b11100110);

        // Variable para hacer el reset
        reset = false;

        return;
}

void main()
{
        // Placa Generica - Implementacion del protocolo
        init();

        // Init del protocol
        initProtocol();

        // FOREVER
        while(true)
        {
                // Hace sus funciones...

                // Protocolo
                runProtocol(&command);
        }

        return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
        int crc, i, len;

        // Calculo del CRC
        crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

        // CRC ok?
        if (cmd->crc != crc)
        {
                // Creo respuesta de error
                response.len = MIN_LENGTH + cmd->len + 2 + 1;
                response.to = cmd->from;
                response.from = THIS_CARD;
                response.cmd = COMMON_ERROR;
                response.data[0] = 0x00;
                // Agrego el paquete que contiene el error de CRC
                response.data[1] = cmd->len;
                response.data[2] = cmd->to;
                response.data[3] = cmd->from;
                response.data[4] = cmd->cmd;
                // Campo data
```

```
                len = cmd->len - MIN_LENGTH;
                for (i = 0; i < len; i++)
                        response.data[5 + i] = (cmd->data)[i];
                // CRC erroneo
                response.data[5 + len] = cmd->crc;
                // CRC esperado
                response.data[5 + len + 1] = crc;
                // CRC de la respuesta
                response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

                crcOK = false;
                return;
        }

        crcOK = true;

        // Minimo todos setean esto
        response.len = MIN_LENGTH;
        response.to = cmd->from & 0x77;
        response.from = THIS_CARD;
        response.cmd = cmd->cmd | 0x80;

        switch (cmd->cmd)
        {
                // Comandos comunes
                case COMMON_INIT:
                        init();
                        // Enviar la descripcion de la placa en texto plano
                        strcpy(response.data, DESC);
                        response.len += strlen(response.data);
                break;
                case COMMON_RESET:
                        // Enviar la descripcion de la placa en texto plano
                        strcpy(response.data, DESC);
                        response.len += strlen(response.data);
                        // Reset!
                        reset = true;
                break;
                case COMMON_PING:
                        // No hace falta hacer mas nada
                break;
                case COMMON_ERROR:
                        // Por ahora se ignora el comando
                break;

                /* Comandos especificos */

                case 0x40:
                        /*
                        :DATO:
                        -
                        :RESP:
                        -
                        */
                break;

                default:
                        response.len++;
                        response.cmd = COMMON_ERROR;
                        response.data[0] = 0x01; // Comando desconocido
                break;
        }

        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

        return;
}
```

B.2. Archivo *protocolo.c*

Código fuente común que genera un buffer cíclico con los paquetes entrantes, parsea el buffer y llama a la función que analiza al paquete según la implementación propia para cada placa.

```
#include <../../protocolo/src/protocol.h>

short reset;
short crcOK;
short sendResponse;

char buffer[MAX_BUFFER_SIZE];
int buffer_write;
int buffer_read;
int data_length;

// Comando parseado
struct command_t command;
// Respuesta
struct command_t response;

// Interrupcion del RS232
#INT_RDA
void RS232()
{
        disable_interrupts(INT_RDA);
        // Un nuevo dato...
        buffer[buffer_write++] = getc();
        data_length++;
        if (buffer_write == MAX_BUFFER_SIZE)
                buffer_write -= MAX_BUFFER_SIZE;
        enable_interrupts(INT_RDA);
        return;
}

/* Envia los datos por el pto serial */
void initProtocol()
{
        // Variables de comunicacion
        buffer_write = 0;
        buffer_read = 0;
        data_length = 0;
        crcOK = false;

        // Interrupcion Rcv
        enable_interrupts(INT_RDA);

        // Habilito las interrupciones
        enable_interrupts(GLOBAL);
}

/* Envia los datos por el pto serial */
void send(struct command_t * cmd)
{
        int i, len;

        len = cmd->len - 4;
        putc(cmd->len);
        putc(cmd->to);
        putc(cmd->from);
        putc(cmd->cmd);

        for (i = 0; i < len; i++)
        {
                putc((cmd->data)[i]);
        }

        // Enviar el CRC
        putc(cmd->crc);

        return;
}
```

```
void runProtocol(struct command_t * cmd)
{
        // Analiza el buffer
        if (buffer[buffer_read] < data_length)
        {
                data_length -= buffer[buffer_read] + 1;

                cmd->len = buffer[buffer_read++];

                if (buffer_read == MAX_BUFFER_SIZE)
                        buffer_read = 0;

                cmd->to = buffer[buffer_read++];

                if (buffer_read == MAX_BUFFER_SIZE)
                        buffer_read = 0;

                cmd->from = buffer[buffer_read++];

                if (buffer_read == MAX_BUFFER_SIZE)
                        buffer_read = 0;

                cmd->cmd = buffer[buffer_read++];

                if (buffer_read == MAX_BUFFER_SIZE)
                        buffer_read = 0;

                // Obtiene el campo DATA
                if ((buffer_read + cmd->len - MIN_LENGTH) > MAX_BUFFER_SIZE)
                {
                        // DATA esta partido en el buffer ciclico
                        memcpy(cmd->data, buffer + buffer_read, MAX_BUFFER_SIZE - buffer_read);
                        memcpy(cmd->data + MAX_BUFFER_SIZE - buffer_read, buffer,
                                cmd->len - MIN_LENGTH - MAX_BUFFER_SIZE + buffer_read);
                } else {
                        // DATA esta continuo
                        memcpy(cmd->data, buffer + buffer_read, cmd->len - MIN_LENGTH);
                }

                buffer_read += cmd->len - MIN_LENGTH;
                if (buffer_read >= MAX_BUFFER_SIZE)
                        buffer_read -= MAX_BUFFER_SIZE;

                cmd->crc = buffer[buffer_read++];

                if (buffer_read == MAX_BUFFER_SIZE)
                        buffer_read = 0;

                sendResponse = true;

                // Soy el destinatario?
                if (cmd->to == THIS_CARD)
                {
                        // Ejecuta el comando
                        doCommand(cmd);
                } else // Es broadcast?
                        if ((cmd->to & 0xF0) == 0xF0)
                {
                        // Ejecuta el comando
                        doCommand(cmd);

                        if (crcOK == true)
                        {
                                // Envia la respuesta
                                send(&response);
                                // Envia nuevamente el comando recibido
                                response = *cmd;
                        }
                } else // Es broadcast para mi grupo?
                        if (((cmd->to & 0x0F) == 0x0F) &&
                                ((cmd->to & 0xF0) == THIS_GROUP))
                {
                        // Ejecuta el comando
```

```
                        doCommand(cmd);
#ifdef RESEND_GROUP_BROADCAST
                        if (crcOK == true)
                        {
                                // Envia la respuesta
                                send(&response);
                                // Envia nuevamente el comando recibido
                                response = *cmd;
                        }
#endif
                } else {
                        response = *cmd;
                }

                // Envia la respuesta?
                if (sendResponse == true)
                {
                        send(&response);
                }

        }

        // Reset del micro
        if (reset == true)
        {
                reset_cpu();
        }

}

int generate_8bit_crc(char* data, int length, int pattern)
{
        // TODO: reemplazar por el crc?

        int crc_byte, i;

        crc_byte = data[0];

        for (i = 1; i < length; i++)
                crc_byte ^= data[i];

        return crc_byte;
}
```

B.3. Placa controladora de motor de dc

Código fuente de la placa controladora de motor de corriente contínua.

```
//CCS PCM V4.023 COMPILER

#define CARD_GROUP      MOTOR_DC        // Ver protocol.h
#define CARD_ID         0               // Valor entre 0 y E

// Descripcion de la placa
#define DESC            "CONTROL MOTOR DC 1.0" // Maximo DATA_SIZE bytes

/* Modulo Motor - main.c
 * PIC16F88 - MAX232 - L298 - MR-2-60-FA
 *
 *                               PIC16F88
 *               .-----------------------------------.
 *          VREF -|RA2/AN2/CVREF/VREF           RA1/AN1|- MOTOR:CHA_B
 *           LED -|RA3/AN3/VREF+/C1OUT          RA0/AN0|- L298:SEN
 *           LED -|RA4/AN4/T0CKI/C2OUT    RA7/OSC1/CLKI|- XT CLOCK pin1, 27pF to GND
 * RST/ICD2:MCLR -|RA5/MCLR/VPP           RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *           GND -|VSS                           VDD|- +5v
 *   L298:ENABLE -|RB0/INT/CCP1       RB7/AN6/PGD/T1OSI|- ICD2:PGD
 *     MOTOR:IDX -|RB1/SDI/SDA  RB6/AN5/PGC/T1OSO/T1CKI|- ICD2:PGC/MOTOR:CHA_A
 *  MAX232:R1OUT -|RB2/SDO/RX/DT           RB5/SS/TX/CK|- MAX232:T1IN
 * L298:INPUT_B/ -|RB3/PGM/CCP1             RB4/SCK/SCL|- L298:INPUT_A
 *     ICD2:PGM   '-----------------------------------'
 *
```

```
 */

#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// MAX232
#bit tx=portb.5
#bit rx=portb.2

// Led
#bit led1=porta.3
#bit led2=porta.4

// L298
#bit inputA=portb.4
#bit inputB=portb.3
#bit enable=portb.0
#bit sensor=porta.0

// Motor inuts
#bit motorIDX=portb.1
#bit channelA=portb.6
#bit channelB=porta.1

#include <../../protocolo/src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE];   // Buffer de recepcion de comandos
int buffer_write;                               // Indice de escritura
int buffer_read;                                // Indice de lectura
int data_length;                                // Largo de los datos en el buffer

struct command_t command;       // Comando parseado
struct command_t response;      // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecula el comando

***/

#define MAX_CONSUMPTION 150
#define MAX_CONSUMPTION_COUNT   5

// Correcion de la cantidad de cuentas por segundo en base al periodo del TMR0
#define INTERVAL_CORRECTION     5

// Girar -> clockwise or unclockwise
// Intercambiar entre el motor derecho y el izquierdo
#define CLOCKWISE       1
#define UNCLOCKWISE     -1

// Sentido de giro del motor
signed int turn;
// Cantidad de overflows del TMR0
```

```
long tmr0_ticks;
// Valor acumulado del ADC - Consumo aprox
long adc_value;
// Ultimo valor del consumo
long last_consumption;
// Valor de duty del PWM
signed long duty;
// Cantidad de cuentas del encoder medidas por intervalo
signed long counts_real;
// Cantidad de cuentas del encoder esperadas por intervalo
signed long counts_expected;
// Cantidad de cuentas del encoder historicas (32 bits)
signed int32 counts_total;
// Cantidad de cuentas del encoder restantes para deterner el motor
signed long counts_to_stop;
// Cantidad de cuentas del encoder desde el ultimo intevalo
signed long last_counts;
signed long last_counts2;
// Tengo una cantidad de cuentas para hacer?
short counts_check;
// Corrijo el PWM segun lo esperado?
short correct_duty;
// Indica que hay que enviar una alarma de consumo
short consumption_alarm;
// Cuenta cuantas alarmas se enviaron
int alarm_count;
// Indica que hay que avisar que el motor se detuvo
short shutdown_alarm;
// Indica si se apagaron los motores por el alto consumo
short motor_shutdown;

// Variables temporales
signed int32 * tmp32;
signed long * tmp16;

/* Setea el PWM */
void SetPWM(signed long pwm);

// Interrupcion del Timer0
#INT_RTCC
void Timer0_INT()
{
        // Seteo el valor para que interrumpa cada 6.25ms
        set_timer0(12);

        // Comienza la lectura del ADC
        read_adc(ADC_START_ONLY);

        // Agrego al historico de cuentas el ultimo acumulado
        counts_real = get_timer1();
        counts_total += (counts_real - last_counts2) * turn;
        last_counts2 = counts_real;

        // Tengo una cantidad de cuentas para hacer?
        if (counts_check == 1)
        {
                // Verifico si pasaron las cuentas que se habian pedido
                if (counts_to_stop < 1)
                {
                        // Detengo el motor
                        counts_check = 0;
                        counts_to_stop = 0;
                        counts_expected = 0;
                        duty = 0;
                        SetPWM(duty);
                        correct_duty = 0;
                        last_counts = 0;
                } else {
                        counts_real = get_timer1();
                        counts_to_stop -= (counts_real - last_counts);// * turn;
                        correct_duty = 1;
                        last_counts = counts_real;
                }
        } else {
```

```
                correct_duty = 1;
        }

        // Tomo la muestra
        adc_value += read_adc(ADC_READ_ONLY);

        if (++tmr0_ticks == 32)
        {
                // Entra cada 200ms

                // Obtengo la cantidad de cuentas desde la ultima entrada
                counts_real = get_timer1();
                set_timer1(0);
                last_counts = 0;
                last_counts2 = 0;
                // Promedio el consumo segun la cantidad de tmr0_ticks
                last_consumption = adc_value / tmr0_ticks;

                if (last_consumption >= MAX_CONSUMPTION)
                {
                        consumption_alarm = 1;
                        if (alarm_count++ == MAX_CONSUMPTION_COUNT)
                        {
                                motor_shutdown = 1;
                                alarm_count = 0;
                                consumption_alarm = 0;
                                shutdown_alarm = 1;
                        }
                }

                tmr0_ticks = 0;

                // Mantengo el consumo promedio desde que arranque y borro el temporal
                adc_value = 0;

                // Corrijo el PWM segun lo esperado
                if ((correct_duty == 1) && (counts_real != counts_expected))
                {
                        duty += (counts_expected - counts_real) * 5;
                        if (duty > 1023L)
                                duty = 1023;
                        else if (duty < 0)
                                duty = 0;
                        SetPWM(duty * turn);
                } else if ((counts_expected == 0) && (duty != 0)) {
                        SetPWM(duty = 0);
                }
        }
        return;
}

void init()
{
        // Inicializa puertos
        set_tris_a(0b11100111);
        set_tris_b(0b11100110);

        // ***ADC***
        setup_port_a(sAN0);//|VSS_VREF);
        setup_adc(ADC_CLOCK_INTERNAL);
        set_adc_channel(0);
        setup_adc_ports(sAN0);
        // Deberia usar VREF_A2... probar
        setup_vref(VREF_HIGH | 8); // VREF a 2.5V -> no hay cambios...

        // ***PWM***
        setup_ccp1(CCP_PWM);
        // Seteo al PWM con f: 4.88 kHz, duty = 0
        set_pwm1_duty(0);
        setup_timer_2(T2_DIV_BY_4, 255, 1);

        // ***TIMER1 - ENCODER COUNTER***
        // Seteo el Timer1 como fuente externa y sin divisor
        setup_timer_1(T1_EXTERNAL | T1_DIV_BY_1);
```

```
        set_timer1(0);

        // ***TIMER0 - TIME BASE***
        // Seteo el Timer0 como clock -> dt = 6.25ms
        setup_counters(RTCC_INTERNAL, RTCC_DIV_128);
        set_timer0(12);
        // Interrupcion sobre el Timer0
        enable_interrupts(INT_RTCC);

        // Habilito las interrupciones
        enable_interrupts(GLOBAL);

        // Variable para hacer el reset
        reset = false;

        // Sentido de giro del motor
        turn = CLOCKWISE;
        // Cantidad de overflows del TMR0
        tmr0_ticks = 0;
        // Valor acumulado del ADC - Consumo aprox
        adc_value = 0;
        last_consumption = 0;
        // Valor de duty del PWM
        duty = 0;
        // Cantidad de cuentas del encoder esperadas por intervalo
        counts_expected = 0;
        // Cantidad de cuentas del encoder totales
        counts_total = 0;
        // Cantidad de cuentas del encoder restantes para deterner el motor
        counts_to_stop = 0;
        // Cantidad de cuentas del encoder desde el ultimo intevalo
        last_counts = 0;
        last_counts2 = 0;
        // Tengo una cantidad de cuentas para hacer?
        counts_check = 0;
        // Corrijo el PWM segun lo esperado?
        correct_duty = 1;
        // Indica que hay que enviar una alarma de consumo
        consumption_alarm = 0;
        // Cuenta cuantas alarmas se enviaron
        alarm_count = 0;
        // Indica que hay que enviar una alarma de consumo
        shutdown_alarm = 0;
        // Indica si se apagaron los motores por el alto consumo
        motor_shutdown = 0;

        return;
}

/* Setea el duty del PWM segun el valor. Positivo o negativo determina el sentido */
void setPWM(signed long pwm)
{
        if (pwm < 0)
        {
                inputA = 0;
                inputB = 1;
        } else {
                inputA = 1;
                inputB = 0;
        }

        if (motor_shutdown == 1)
        {
                pwm = 0;
        } else {
                pwm = (abs(pwm));

                if (pwm > 1023L)
                        pwm = 1023;
        }

        set_pwm1_duty(pwm);

        return;
```

```
}

void main()
{
        // Placa Generica - Implementacion del protocolo
        init();

        // Init del protocol
        initProtocol();

counts_expected = 30;

        // FOREVER
        while(true)
        {
                // Hace sus funciones -> interrupcion

                // Enviar alarma de alto consumo
                if (consumption_alarm == 1)
                {
                        /* Indica al controlador principal que hay un consumo extremo en el motor,
                        posiblemente un atasco del motor o de la rueda.
                        :DATO:
                        Numero entero positivo de 16 bits en el rango desde 0x0000 hasta
                        0x03FF, que representa el consumo ante el que sono la alarma.
                        :RESP:
                        -
                        */
                        command.len = MIN_LENGTH + 2;
                        command.to = MAIN_CONTROLLER;
                        command.from = THIS_CARD;
                        command.cmd = DC_MOTOR_MOTOR_STRESS_ALARM;
                        // A la posicion 0 dentro de command.data la tomo como signed long *
                        tmp16 = (command.data);
                        // Le asigno el valor del ultimo consumo del motor
                        (*tmp16) = last_consumption;
                        command.crc = generate_8bit_crc((char *)(&command), command.len, CRC_PATTERN);
                        consumption_alarm = 0;
                        // Envio del comando
                        send(&command);
                }

                // Enviar aviso de motor apagado
                if (shutdown_alarm == 1)
                {
                        /* Indica al controlador principal que el motor ha sido apagado debido al alto
                        consumo. Enviado luego de sucesivos avisos del comando DC_MOTOR_MOTOR_STRESS_ALARM.
                        :DATO:
                        Numero entero positivo de 16 bits en el rango desde 0x0000 hasta
                        0x03FF, que representa el consumo ante el que sono la alarma.
                        :RESP:
                        -
                        */
                        command.len = MIN_LENGTH + 2;
                        command.to = MAIN_CONTROLLER;
                        command.from = THIS_CARD;
                        command.cmd = DC_MOTOR_MOTOR_SHUT_DOWN_ALARM;
                        // A la posicion 0 dentro de response->data la tomo como signed long *
                        tmp16 = (command.data);
                        // Le asigno el valor del ultimo consumo del motor
                        (*tmp16) = last_consumption;
                        command.crc = generate_8bit_crc((char *)(&command), command.len, CRC_PATTERN);
                        shutdown_alarm = 0;
                        // Envio del comando
                        send(&command);
                }

                // Protocolo
                runProtocol(&command);
        }

        return;
}
```

```
/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
        int crc, i, len;

        // Calculo del CRC
        crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

        // CRC ok?
        if (cmd->crc != crc)
        {
                // Creo respuesta de error
                response.len = MIN_LENGTH + cmd->len + 2 + 1;
                response.to = cmd->from;
                response.from = THIS_CARD;
                response.cmd = COMMON_ERROR;
                response.data[0] = 0x00;
                // Agrego el paquete que contiene el error de CRC
                response.data[1] = cmd->len;
                response.data[2] = cmd->to;
                response.data[3] = cmd->from;
                response.data[4] = cmd->cmd;
                // Campo data
                len = cmd->len - MIN_LENGTH;
                for (i = 0; i < len; i++)
                        response.data[5 + i] = (cmd->data)[i];
                // CRC erroneo
                response.data[5 + len] = cmd->crc;
                // CRC esperado
                response.data[5 + len + 1] = crc;
                // CRC de la respuesta
                response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

                crcOK = false;
                return;
        }

        crcOK = true;

        // Minimo todos setean esto
        response.len = MIN_LENGTH;
        response.to = cmd->from & 0x77;
        response.from = THIS_CARD;
        response.cmd = cmd->cmd | 0x80;

        switch (cmd->cmd)
        {
                // Comandos comunes
                case COMMON_INIT:
                        init();
                        // Enviar la descripcion de la placa en texto plano
                        strcpy(response.data, DESC);
                        response.len += strlen(response.data);
                break;
                case COMMON_RESET:
                        // Enviar la descripcion de la placa en texto plano
                        strcpy(response.data, DESC);
                        response.len += strlen(response.data);
                        // Reset!
                        reset = true;
                break;
                case COMMON_PING:
                        // No hace falta hacer mas nada
                break;
                case COMMON_ERROR:
                        // Por ahora se ignora el comando
                break;

                /* Comandos especificos */

                case DC_MOTOR_SET_DIRECTION:
                        /* Seteo del sentido de giro del motor
                        :DATO:
                        0x00 para sentido horario o 0x01 para sentido anti-horario.
```

```
        :RESP:
        -
        */
        if (((cmd->data)[0] & 0x01) == 0)
        {
                turn = CLOCKWISE;
        } else {
                turn = UNCLOCKWISE;
        }
break;
case DC_MOTOR_SET_DC_SPEED:
        /* Seteo de la velocidad del motor en cuentas del encoder por segundo
        :DATO:
        0x00 para sentido horario o 0x01 para sentido anti-horario. Numero
        entero de 16 bits con signo, que representa la velocidad en cuentas
        por segundos.
        :RESP:
        -
        */
        if (((cmd->data)[0] & 0x01) == 0)
        {
                turn = CLOCKWISE;
        } else {
                turn = UNCLOCKWISE;
        }
        // A la posicion 1 dentro de cmd->data la tomo como signed long *
        tmp16 = (cmd->data) + 1;
        // Le asigno el valor de la velocidad ajustada a 1 segundo
        counts_expected = (*tmp16) / INTERVAL_CORRECTION;
        // Habilita el motor
        motor_shutdown = 0;
break;
case DC_MOTOR_SET_ENCODER:
        /* Seteo de la cantidad de cuentas historicas del encoder
        :DATO:
        Numero entero de 32 bits con signo, con el valor para setear en el
        historico del encoder.
        :RESP:
        -
        */
        // A la posicion 0 dentro de cmd->data la tomo como signed int32 *
        tmp32 = (cmd->data);
        // Le asigno el valor de las cuentas historicas
        counts_total = (*tmp32);
break;
case DC_MOTOR_GET_ENCODER:
        /* Obtener la cantidad de cuentas historicas del encoder
        :DATO:
        -
        :RESP:
        Numero entero de 32 bits con signo, que representa el valor historico
        del encoder.
        */
        // A la posicion 0 dentro de response->data la tomo como signed int32 *
        tmp32 = (response.data);
        // Le asigno el valor de las cuentas historicas
        (*tmp32) = counts_total;
        // Corrijo el largo del paquete
        response.len += 4;
break;
case DC_MOTOR_RESET_ENCODER:
        /* Resetear las cuentas historicas a cero
        :DATO:
        -
        :RESP:
        -
        */
        counts_total = 0;
break;
case DC_MOTOR_SET_ENCODER_TO_STOP:
        /* Seteo de cuantas cuentas debe girar hasta detenerse
        :DATO:
        Numero entero de 16 bits con signo, que representa la cantidad de
        cuentas del encoder restantes para que el motor se detenga.
```

```
        :RESP:
        -
        */
        // A la posicion 0 dentro de cmd->data la tomo como signed long *
        tmp16 = (cmd->data);
        // Le asigno el valor de la velocidad ajustada a 1 segundo
        counts_to_stop = (*tmp16);
        // Habilito el chequeo de cuentas para detener el motor
        counts_check = 1;
break;
case DC_MOTOR_GET_ENCODER_TO_STOP:
        /* Obtener la cantidad de las cuentas restantes que quedan por
        realizar hasta detenerse.
        :DATO:
        -
        :RESP:
        Numero entero de 16 bits con signo, que representa la cantidad
        de cuentas del encoder restantes para detener el motor.
        */
        // A la posicion 0 dentro de response->data la tomo como signed long *
        tmp16 = (response.data);
        // Le asigno el valor de la velocidad ajustada a 1 segundo
        (*tmp16) = counts_to_stop;
        // Corrijo el largo del paquete
        response.len += 2;
break;
case DC_MOTOR_DONT_STOP:
        /* Deshace los comandos DC_MOTOR_DONT_STOP y DC_MOTOR_GET_ENCODER_TO_STOP,
        deshabilita el conteo de cuentas para frenar y sigue en el estado actual.
        :DATO:
        -
        :RESP:
        -
        */
        counts_check = 0;
break;
case DC_MOTOR_MOTOR_CONSUMPTION:
        /* Numero entero positivo de 16 bits en el rango desde 0x0000 hasta
        0x03FF, que representa el consumo promedio del ultimo segundo.
        :DATO:
        -
        :RESP:
        -
        */
        // A la posicion 0 dentro de response->data la tomo como signed long *
        tmp16 = (response.data);
        // Le asigno el valor del ultimo consumo del motor
        (*tmp16) = last_consumption;
        // Corrijo el largo del paquete
        response.len += 2;
break;
/*case DC_MOTOR_MOTOR_STRESS_ALARM:
break;
case DC_MOTOR_MOTOR_SHUT_DOWN_ALARM:
break;*/
case DC_MOTOR_GET_DC_SPEED:
        /* Obtiene la velocidad del motor en cuentas del encoder por segundo
        :DATO:
        0x00 para sentido horario o 0x01 para sentido anti-horario. Numero
        entero de 16 bits con signo, que representa la velocidad en cuentas
        por segundos.
        :RESP:
        -
        */
        // Sentido de giro del motor
        if (turn == CLOCKWISE)
        {
                (response.data)[0] = 0x00;
        } else {
                (response.data)[0] = 0x01;
        }
        // A la posicion 1 dentro de response->data la tomo como signed long *
        tmp16 = (response.data) + 1;
        // Le asigno el valor de la velocidad ajustada a 1 segundo
```

```
                        (*tmp16) = counts_real * INTERVAL_CORRECTION;
                        // Corrijo el largo del paquete
                        response.len += 2;
                break;

                default:
                        response.len++;
                        response.cmd = COMMON_ERROR;
                        response.data[0] = 0x01; // Comando desconocido
                break;
        }

        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

        return;
}
```

B.4. Placa controladora de sensores

Código fuente de la placa controladora de sensores.

```
//CCS PCM V4.023 COMPILER

#define CARD_GROUP      DISTANCE_SENSOR // Ver protocol.h
#define CARD_ID         1               // Valor entre 0 y E

// ID:0 -> Placa de telemetros
// ID:1 -> Placa de sensores de piso y sensor de ultrasonido
// ID:2 -> Placa de telemetros

// Descripcion de la placa
#define DESC            "PLACA SENSORES 1.0" // Maximo DATA_SIZE bytes

// Determina el tipo de sensores principales en la placa - CAMBIARLO SEGUN CORRESPONDA
// Posibles valores: FLOOR_SENSORS o TELEMETERS_SENSORS o NONE
//#define SENSORS_TYPE  TELEMETERS_SENSORS

// Determina contra que esta conectado el pin TRIGGER - CAMBIARLO SEGUN CORRESPONDA
// Posibles valores: NONE o ULTRASONIC_SENSOR o SWITCH_SENSOR o LED
//#define TRIGGER_TYPE  ULTRASONIC_SENSOR
//#define TRIGGER_TYPE  SWITCH_SENSOR

#if CARD_ID == 0
        #define SENSORS_TYPE    TELEMETERS_SENSORS
        #define TRIGGER_TYPE    SWITCH_SENSOR
        #define DEFAULT_MASK    0x2F
#elif CARD_ID == 1
        #define SENSORS_TYPE    FLOOR_SENSORS
        #define TRIGGER_TYPE    ULTRASONIC_SENSOR
        #define DEFAULT_MASK    0x37
#elif CARD_ID == 2
        #define SENSORS_TYPE    TELEMETERS_SENSORS
        #define TRIGGER_TYPE    SWITCH_SENSOR
        #define DEFAULT_MASK    0x2F
#endif

// Define el estado logico para prender o apagar los sensores - CAMBIARLO SEGUN CORRESPONDA
#define SENSOR_ON       0
#define SENSOR_OFF      1

#define SAMPLES_DEFAULT 5

// Ancho del pulso que se debe enviar al sensor de ultrasonido como INIT - CAMBIARLO SEGUN CORRESPONDA
#define ULTRASONIC_INIT_PULSE_WIDTH_US  15

// Frame de muestreo. Cada este tiempo se realiza una nueva lectura ver CalculoTMR.xls
#define COMPLEMENT_TIME                                 12411 // 85ms
// Tiempo necesario para realizar la lectura en los telemetros - CAMBIARLO SEGUN CORRESPONDA
#define TELEMETERS_WAITING_TIME_CYCLES  26792 //62ms
#define TELEMETERS_COMPLEMENT_TIME              51161 //23ms complemento a COMPLEMENT_TIME
// Tiempo necesario para realizar la lectura en los sensores de piso - CAMBIARLO SEGUN CORRESPONDA
```

```
#define FLOOR_WAITING_TIME_CYCLES               62411 //5ms
#define FLOOR_COMPLEMENT_TIME                   15536 //80ms complemento a COMPLEMENT_TIME
// Tiempo necesario para realizar la lectura del sensor de ultrasonido - CAMBIARLO SEGUN CORRESPONDA
#define ULTRASONIC_WAITING_TIME_CYCLES  45536 //32ms
#define ULTRASONIC_COMPLEMENT_TIME              32411 //53ms complemento a COMPLEMENT_TIME

// Tiempo entre el cambio de canal del ADC y una muestra estable (minimo 10us) - CAMBIARLO SEGUN CORRESPONDA
#define ADC_DELAY       20

/* Modulo Generico - main.c
 * PIC16F88 - MAX232 - SENSORES
 *
 *                              PIC16F88
 *                .-----------------------------------.
 *       SENSE_3 -|RA2/AN2/CVREF/VREF           RA1/AN1|- SENSE_2
 *       SENSE_4 -|RA3/AN3/VREF+/C1OUT          RA0/AN0|- SENSE_1
 *       SENSE_5 -|RA4/AN4/TOCKI/C2OUT    RA7/OSC1/CLKI|- XT CLOCK pin1, 27pF to GND
 * RST/ICD2:MCLR -|RA5/MCLR/VPP           RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *           GND -|VSS                         VDD|- +5v
 *       TRIGGER -|RB0/INT/CCP1       RB7/AN6/PGD/T1OSI|- ICD2:PGD/SENSOR_5
 *      SENSOR_1 -|RB1/SDI/SDA  RB6/AN5/PGC/T1OSO/T1CKI|- ICD2:PGC/SENSOR_4
 *  MAX232:R1OUT -|RB2/SDO/RX/DT           RB5/SS/TX/CK|- MAX232:T1IN
 *      SENSOR_2 -|RB3/PGM/CCP1             RB4/SCK/SCL|- SENSOR_3
 *                '-----------------------------------'
 *
 */

#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// MAX232
#bit tx=portb.5
#bit rx=portb.2

// Pin para el control del sensore de ultrasonido, switch on/off o led
#bit trigger=portb.0
// Led
#bit led1=portb.0
// Pin para switch On/Off
#bit inOnOff=portb.0
// Pin de control del sensor de ultrasonido
#bit ultrasonido=portb.0

// Sensores - Base de los transistores
#bit sensor1=portb.1
#bit sensor2=portb.3
#bit sensor3=portb.4
#bit sensor4=portb.6
#bit sensor5=portb.7

// Sensores - Salida de los sensores (SENSE)
#bit sense1=porta.0
#bit sense2=porta.1
#bit sense3=porta.2
#bit sense4=porta.3
#bit sense5=porta.4

// Estados para la maquina de estados que controla las funciones de la placa
#define STATE_FREE                      0
#define STATE_START_READING     1
#define STATE_WAIT_TO_READ      2
```

```
#define STATE_READ_VALUES       3
#define STATE_WAITING           4

// Estados para la lectura del sensor de ultrasonido
#define USONIC_STATE_START      0
#define USONIC_STATE_STOP       1

// Posibles conexiones del pin TRIGGER
#define NONE                      (0x00)
#define ULTRASONIC_SENSOR (NONE + 1)
#define SWITCH_SENSOR     (ULTRASONIC_SENSOR + 1)
#define LED                               (SWITCH_SENSOR + 1)

// Posibles sensores principales
#define TELEMETERS_SENSORS (NONE + 1)
#define FLOOR_SENSORS      (TELEMETERS_SENSORS + 1)

// Tiempo maximo para NONE
#define NONE_TIME       65530 //0ms
#define SWITCH_TIME 65530 //0ms
#define LED_TIME        65530 //0ms

#if SENSORS_TYPE == FLOOR_SENSORS
        #if TRIGGER_TYPE == ULTRASONIC_SENSOR
                #define SENSORS_WAITING_TIME ULTRASONIC_WAITING_TIME_CYCLES
                #define WAITING_TIME ULTRASONIC_COMPLEMENT_TIME //Complemento 85ms -> 53ms
        #else
                #define SENSORS_WAITING_TIME FLOOR_WAITING_TIME_CYCLES
                #define WAITING_TIME FLOOR_COMPLEMENT_TIME //Complemento 85ms -> 80ms
        #endif
#elif SENSORS_TYPE == TELEMETERS_SENSORS
        #define SENSORS_WAITING_TIME TELEMETERS_WAITING_TIME_CYCLES
        #define WAITING_TIME TELEMETERS_COMPLEMENT_TIME //Complemento 85ms -> 23ms
#elif SENSORS_TYPE == NONE
        #if TRIGGER_TYPE == ULTRASONIC_SENSOR
                #define SENSORS_WAITING_TIME ULTRASONIC_WAITING_TIME_CYCLES
                #define WAITING_TIME ULTRASONIC_COMPLEMENT_TIME //Complemento 85ms -> 53ms
        #else
                #define SENSORS_WAITING_TIME NONE_TIME
                #define WAITING_TIME COMPLEMENT_TIME //85ms
        #endif
#endif

#include <../../protocolo/src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE];   // Buffer de recepcion de comandos
int buffer_write;                               // Indice de escritura
int buffer_read;                                // Indice de lectura
int data_length;                                // Largo de los datos en el buffer

struct command_t command;       // Comando parseado
struct command_t response;      // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecula el comando

***/

// IO como entrada o salida
int trisB_value = 0b00100101;

// Determina el estado actual
int state;

// Vector donde se almacenan los valores de los sensores
unsigned long values[6];
```

```
// Cantidad de muestras a tomar por sensor
int samples;

// Mascara que ignora ciertos sensores
int sensorMask;

// Determina cuales sensores estan siendo leidos en este instante
int actualReadSensor;

// Determina cuales son los sensores de los que se pide la lectura
int readSensor;

// Bufferea el pedido de sensores de los que se pide la lectura
int bufferedReadSensor;

// Determina el destinatario actual
int actalTO;

// Informa quien hizo el pedido
int requestFrom;

// Informa de quien vino el ultimo pedido (mientras habia una lectura en curso)
int bufferedFrom;

// Determina el comando actual
int actalCmd;

// Informa cual fue el comando del pedido
int requestCmd;

// Informa cual fue el comando del ultimo pedido (mientras habia una lectura en curso)
int bufferedCmd;

// Estado para el sensor de ultrasonido
short usonic_state;

// Tiempo para el comienzo del pulso del sensor de ultrasonido
unsigned long pulseStart;

// Flag de aviso sobre la interrupcion en TIMER1
short intTMR;

// Tipo de alarma en trigger
int alarmType;

// Flag de alarma en trigger
short triggerAlarm;

/* Habilita los sensores segun corresponda para comenzar la lectura */
void startReading(int sensors);

/* Realiza la lectura sobre los sensores segun corresponda */
void readSensors(int sensors);

/* Interrupcion del TIMER1 */
void int_timer(void);

/* Interrupcion de RB0 */
void int_trigger(void);

/* Crea la respuesta sobre la lectura de los sensores */
void sendValues(int to, int cmd, long * values, int sensors);

/* Interrupcion del TIMER1 */
#INT_TIMER1
void int_timer(void)
{
        // Habilita el flag de aviso que sucedio la interrupcion
        intTMR = 1;
        disable_interrupts(INT_TIMER1);
}

/* Interrupcion de RB0 */
```

```
#INT_EXT
void int_trigger(void)
{
#if TRIGGER_TYPE == ULTRASONIC_SENSOR
        if (usonic_state == USONIC_STATE_START)
        {
                // Tomo el tiempo en que comienza el pulso
                pulseStart = get_timer1();
                // Cambio el tipo de flanco
                ext_int_edge(H_TO_L);
                // Cambio el estado
                usonic_state = USONIC_STATE_STOP;
        } else {
                // Tomo el tiempo y guardo el valor
                values[5] = (get_timer1() - pulseStart)/2;
                // Deshabilita la interupcion
                disable_interrupts(INT_EXT);
        }
#elif TRIGGER_TYPE == SWITCH_SENSOR
        triggerAlarm = 1;
#endif


}

void init()
{
        // Inicializa puertos
        set_tris_a(0b11111111);
        set_tris_b(trisB_value);

        // ***ADC***
        setup_port_a(sAN0);
        setup_adc(ADC_CLOCK_INTERNAL);
        set_adc_channel(0);
        setup_adc_ports(sAN0);
        setup_vref(VREF_HIGH | 8);

        // Seteo el Timer1 como fuente interna
        setup_timer_1(T1_INTERNAL | T1_DIV_BY_8);
        set_timer1(26786);

        // Interrupcion sobre el Timer1
        enable_interrupts(INT_TIMER1);

        // Seteo el pin RB0 - Sensor de ultrasonido :)
        ext_int_edge(L_TO_H);
        enable_interrupts(INT_EXT);

        // Habilito las interrupciones
        enable_interrupts(GLOBAL);

        // Variable para hacer el reset
        reset = false;

        // Apaga todos los sensores
        sensor1 = SENSOR_OFF;
        sensor2 = SENSOR_OFF;
        sensor3 = SENSOR_OFF;
        sensor4 = SENSOR_OFF;
        sensor5 = SENSOR_OFF;

        // Inicializa los valores
        values[0] = 0x0000;
        values[1] = 0x0000;
        values[2] = 0x0000;
        values[3] = 0x0000;
        values[4] = 0x0000;
        values[5] = 0x0000;

        // Muestras iniciales
        samples = SAMPLES_DEFAULT;

        // Inicializa la mascara -> todos habilitados (0x3F)
```

```
        sensorMask = DEFAULT_MASK;

        //Determina el estado actual
        state = STATE_FREE;

        // Sin lectura temprana
        readSensor = 0x00;
        bufferedReadSensor = 0x00;
        actalTO = 0x00;
        requestFrom = 0x00;
        bufferedFrom = 0x00;
        actalCmd = 0x00;
        requestCmd = 0x00;
        bufferedCmd = 0x00;

        alarmType = 0x00;
        triggerAlarm = 0;
#if TRIGGER_TYPE == SWITCH_SENSOR
        disable_interrupts(INT_EXT);
#endif

#if TRIGGER_TYPE == LED
        // TRIGGER como escritura
        bit_clear(trisB_value, 0);
        set_tris_b(trisB_value);
#endif

        return;
}

void sendAlarm()
{
        /*TODO: revisar protocolo*/
        triggerAlarm = 0;
        return;
}

void main()
{
        // Placa Generica - Implementacion del protocolo
        init();

        // Init del protocol
        initProtocol();

        // FOREVER
        while(true)
        {

                // Ejecucion de la maquina de estados
                switch (state)
                {
                        case STATE_FREE:
                                /*
                                * Analiza los paquetes y retransmite o lanza el pedido de lectura
                                */

#if TRIGGER_TYPE == SWITCH_SENSOR
                                // Enviar alarma de trigger
                                if (triggerAlarm == 1)
                                        sendAlarm();
#endif

                                // Protocolo
                                runProtocol(&command);

                                if (readSensor != 0x00)
                                {
                                        // Almacena el pedido sobre los sensores
                                        actualReadSensor = readSensor;
                                        readSensor = 0x00;
                                        actalTO = requestFrom;
                                        actalCmd = requestCmd;
                                        // Cambio de estado
```

```
                                        state = STATE_START_READING;
                                }
                                break;
                        case STATE_START_READING:
                                /*
                                * Genera un pedido de lectura a los sensores que lo necesiten
                                * y setea los TIMERs para contabilizar los tiempos
                                */

                                // Limpio la variable
                                bufferedReadSensor = 0x00;

                                // Inhabilita los sensores enmascarados
                                actualReadSensor &= sensorMask;

                                // Pedido de -START-
                                startReading(actualReadSensor);

                                // Flag de interrupcion
                                intTMR = 0;

                                // Seteo de TIMERs
                                set_timer1(SENSORS_WAITING_TIME);
                                enable_interrupts(INT_TIMER1);

                                // Cambio de estado
                                state = STATE_WAIT_TO_READ;

                                break;
                        case STATE_WAIT_TO_READ:
                                /*
                                * Espera el tiempo necesario para tomar las muestras en los sensores
                                * Recibe nuevos pedidos y los agraga para una proxima lectura
                                */

#if TRIGGER_TYPE == SWITCH_SENSOR
                                // Enviar alarma de trigger
                                if (triggerAlarm == 1)
                                        sendAlarm();
#endif

                                // Protocolo
                                runProtocol(&command);

                                // Almacena el pedido sobre los sensores si no hay otro ya
                                if ((readSensor != 0x00) && (bufferedReadSensor == 0x00))
                                {
                                        bufferedReadSensor = readSensor;
                                        bufferedFrom = requestFrom;
                                        bufferedCmd = requestCmd;
                                        readSensor = 0x00;
                                }

                                // Cambio de estado?
                                if (intTMR == 1)
                                {
                                        // El TIMER1 hizo timeout -> leer sensores
                                        state = STATE_READ_VALUES;
                                }

                                break;
                        case STATE_READ_VALUES:
                                /*
                                * Toma las muestras en los sensores y envia el paquete de respuesta
                                */

                                // Toma las muestras de los sensores
                                readSensors(actualReadSensor);

                                // Mandar paquete de respuesta
                                sendValues(actalTO, actalCMD, values, actualReadSensor);

                                // Flag de interrupcion
                                intTMR = 0;
```

```
                                // Setea el tiempo de espera y pasa al estado de espera
                                set_timer1(WAITING_TIME);
                                enable_interrupts(INT_TIMER1);

                                state = STATE_WAITING;

                                break;
                        case STATE_WAITING:
                                /*
                                * Espera a que pase el tiempo de espera entre lecturas para evitar rebotes
                                * Recibe nuevos pedidos y los agraga para una proxima lectura
                                */

#if TRIGGER_TYPE == SWITCH_SENSOR
                                // Enviar alarma de trigger
                                if (triggerAlarm == 1)
                                        sendAlarm();
#endif

                                // Protocolo
                                runProtocol(&command);

                                // Almacena el pedido sobre los sensores si no hay otro ya
                                if ((readSensor != 0x00) && (bufferedReadSensor == 0x00))
                                {
                                        bufferedReadSensor = readSensor;
                                        bufferedFrom = requestFrom;
                                        bufferedCmd = requestCmd;
                                        readSensor = 0x00;
                                }

                                // Cambio de estado?
                                if (intTMR == 1)
                                {
                                        // El TIMER1 hizo timeout
                                        if (bufferedReadSensor != 0x00)
                                        {
                                                // Hay un pedido pendiente y lo carga
                                                actualReadSensor = bufferedReadSensor;
                                                bufferedReadSensor = 0x00;
                                                actalTO = bufferedFrom;
                                                actalCmd = bufferedCmd;
                                                // Cambio de estado
                                                state = STATE_START_READING;
                                        } else {
                                                state = STATE_FREE;
                                        }
                                }

                                break;
                        default:
                                init();
                                break;
                }
        }

        return;
}

/* Habilita los sensores segun corresponda para comenzar la lectura*/
void startReading(int sensors)
{
        // Sensor1
        if (bit_test(sensors, 0) == 1)
                sensor1 = SENSOR_ON;

        // Sensor2
        if (bit_test(sensors, 1) == 1)
                sensor2 = SENSOR_ON;

        // Sensor3
        if (bit_test(sensors, 2) == 1)
                sensor3 = SENSOR_ON;
```

```
        // Sensor4
        if (bit_test(sensors, 3) == 1)
                sensor4 = SENSOR_ON;

        // Sensor5
        if (bit_test(sensors, 4) == 1)
                sensor5 = SENSOR_ON;

        // Sensor6
#if TRIGGER_TYPE == ULTRASONIC_SENSOR
        // Sensor6 -> ULTRASONIC_SENSOR
        if (bit_test(sensors, 5) == 1)
        {
                // Comienza el pulso de habilitacion -> TRIGGER como escritura
                bit_clear(trisB_value, 0);
                set_tris_b(trisB_value);
                // Pin en estado habilitado -> envio del pulso INIT
                trigger = 1;
                delay_us(ULTRASONIC_INIT_PULSE_WIDTH_US);
                trigger = 0;
                // Termina el pulso de habilitacion -> TRIGGER como lectura
                bit_set(trisB_value, 0);
                set_tris_b(trisB_value);
                // Setea la interrupcion sobre RB0 en flanco ascendente
                ext_int_edge(L_TO_H);
                // Seteo el estado actual del pulso del sensor de ultrasonido
                usonic_state = USONIC_STATE_START;
                // Habilita la interrupcion
                enable_interrupts(INT_EXT);
        }
#elif TRIGGER_TYPE == SWITCH_SENSOR
        // Sensor6 -> SWITCH_SENSOR
        if (bit_test(sensors, 5) == 1)
        {
                if (trigger == 1)
                        values[5] = 0xFFFF;
                else
                        values[5] = 0;
        }
#endif

        return;
}

/* Realiza la lectura sobre los sensores segun corresponda */
void readSensors(int sensors)
{
        int i;

        values[0] = 0;
        values[1] = 0;
        values[2] = 0;
        values[3] = 0;
        values[4] = 0;

        for (i = 0; i < samples; i++)
        {
                // Sensor1
                if (bit_test(sensors, 0) == 1)
                {
                        // ADC en el pin correcto
                        set_adc_channel(0);
                        // Espera el tiempo necesario
                        delay_us(ADC_DELAY);
                        // Toma la muestra
                        values[0] += read_adc();
                }

                // Sensor2
                if (bit_test(sensors, 1) == 1)
                {
                        // ADC en el pin correcto
                        set_adc_channel(1);
```

```
                        // Espera el tiempo necesario
                        delay_us(ADC_DELAY);
                        // Toma la muestra
                        values[1] += read_adc();
                }

                // Sensor3
                if (bit_test(sensors, 2) == 1)
                {
                        // ADC en el pin correcto
                        set_adc_channel(2);
                        // Espera el tiempo necesario
                        delay_us(ADC_DELAY);
                        // Toma la muestra
                        values[2] += read_adc();
                }

                // Sensor4
                if (bit_test(sensors, 3) == 1)
                {
                        // ADC en el pin correcto
                        set_adc_channel(3);
                        // Espera el tiempo necesario
                        delay_us(ADC_DELAY);
                        // Toma la muestra
                        values[3] += read_adc();
                }

                // Sensor5
                if (bit_test(sensors, 4) == 1)
                {
                        // ADC en el pin correcto
                        set_adc_channel(4);
                        // Espera el tiempo necesario
                        delay_us(ADC_DELAY);
                        // Toma la muestra
                        values[4] += read_adc();
                }
        }

        /*sensor1 = SENSOR_OFF;
        sensor2 = SENSOR_OFF;
        sensor3 = SENSOR_OFF;
        sensor4 = SENSOR_OFF;
        sensor5 = SENSOR_OFF;*/

        values[0] /= samples;
        values[1] /= samples;
        values[2] /= samples;
        values[3] /= samples;
        values[4] /= samples;

        return;
}

/* Crea la respuesta sobre la lectura de los sensores */
void sendValues(int to, int cmd, long * values, int sensors)
{
        int idx = 1, i;
        signed long * tmp16;

        command.len = MIN_LENGTH + 1;
        command.to = to;
        command.from = THIS_CARD;
        command.cmd = cmd;
        command.data[0] = sensors;

        // Valores de los sensores en command.data segun corresponda
        for (i = 0; i < 6; i++)
        {
                if (bit_test(sensors, i) == 1)
                {
                        // A la posicion 0 dentro de command.data la tomo como signed long *
                        tmp16 = (command.data + idx);
```

```
                        // Le asigno el valor del sensor
                        (*tmp16) = values[i];
                        idx+=2;
                        command.len += 2;
                }
        }

        command.crc = generate_8bit_crc((char *)(&command), command.len, CRC_PATTERN);
        // Envio del comando
        send(&command);

        return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
        int crc, i, len;

        // Calculo del CRC
        crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

        // CRC ok?
        if (cmd->crc != crc)
        {
                // Creo respuesta de error
                response.len = MIN_LENGTH + cmd->len + 2 + 1;
                response.to = cmd->from;
                response.from = THIS_CARD;
                response.cmd = COMMON_ERROR;
                response.data[0] = 0x00;
                // Agrego el paquete que contiene el error de CRC
                response.data[1] = cmd->len;
                response.data[2] = cmd->to;
                response.data[3] = cmd->from;
                response.data[4] = cmd->cmd;
                // Campo data
                len = cmd->len - MIN_LENGTH;
                for (i = 0; i < len; i++)
                        response.data[5 + i] = (cmd->data)[i];
                // CRC erroneo
                response.data[5 + len] = cmd->crc;
                // CRC esperado
                response.data[5 + len + 1] = crc;
                // CRC de la respuesta
                response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

                crcOK = false;
                return;
        }

        crcOK = true;

        // Minimo todos setean esto
        response.len = MIN_LENGTH;
        response.to = cmd->from & 0x77;
        response.from = THIS_CARD;
        response.cmd = cmd->cmd | 0x80;

        switch (cmd->cmd)
        {
                // Comandos comunes
                case COMMON_INIT:
                        init();
                        // Enviar la descripcion de la placa en texto plano
                        strcpy(response.data, DESC);
                        response.len += strlen(response.data);
                break;
                case COMMON_RESET:
                        // Enviar la descripcion de la placa en texto plano
                        strcpy(response.data, DESC);
                        response.len += strlen(response.data);
                        // Reset!
                        reset = true;
```

```
                break;
                case COMMON_PING:
                        // No hace falta hacer mas nada
                break;
                case COMMON_ERROR:
                        // Por ahora se ignora el comando
                break;

                /* Comandos especificos */

                case DISTANCE_SENSOR_ON_DISTANCE_SENSOR:
                        /* Enciende el sensor de distancia indicado.
                        :DATO:
                        Valor de 0x00 a 0x05 que representa el ID del sensor a encender.
                        El ID 0x05 hace referencia al led de la placa si esta presente,
                        en caso contrario se ignora.
                        :RESP:
                        -
                        */
                        // Enciende los sensores segun corresponda
                        if ((cmd->data)[0] == 0)
                                // Sensor1
                                sensor1 = SENSOR_ON;
                        else if ((cmd->data)[0] == 1)
                                // Sensor2
                                sensor2 = SENSOR_ON;
                        else if ((cmd->data)[0] == 2)
                                // Sensor3
                                sensor3 = SENSOR_ON;
                        else if ((cmd->data)[0] == 3)
                                // Sensor4
                                sensor4 = SENSOR_ON;
                        else if ((cmd->data)[0] == 4)
                                // Sensor5
                                sensor5 = SENSOR_ON;
#if TRIGGER_TYPE == LED
                        else if ((cmd->data)[0] == 5)
                                // Led
                                led1 = 1;
#endif
                break;
                case DISTANCE_SENSOR_OFF_DISTANCE_SENSOR:
                        /* Apaga el sensor de distancia indicado.
                        :DATO:
                        Valor de 0x00 a 0x05 que representa el ID del sensor a apagar. El
                        ID 0x05 hace referencia al sensor de ultrasonido o switch de la placa.
                        :RESP:
                        -
                        */
                        // Apaga los sensores segun corresponda
                        if ((cmd->data)[0] == 0)
                                // Sensor1
                                sensor1 = SENSOR_OFF;
                        else if ((cmd->data)[0] == 1)
                                // Sensor2
                                sensor2 = SENSOR_OFF;
                        else if ((cmd->data)[0] == 2)
                                // Sensor3
                                sensor3 = SENSOR_OFF;
                        else if ((cmd->data)[0] == 3)
                                // Sensor4
                                sensor4 = SENSOR_OFF;
                        else if ((cmd->data)[0] == 4)
                                // Sensor5
                                sensor5 = SENSOR_OFF;
#if TRIGGER_TYPE == LED
                        else if ((cmd->data)[0] == 5)
                                // Led
                                led1 = 0;
#endif
                break;
                case DISTANCE_SENSOR_SET_MASK:
                        /* Habilita o deshabilita cada uno de los sensores de distancia conectados al
                        controlador. Permite identificar los sensores a los que se debera tener en
```

```
        cuenta para futuras lecturas.
        :DATO:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor a
        habilitar o deshabilitar. Si 2^ID = 1 entonces el sensor ID esta habilitado.
        Si 2^ID = 0 entonces el sensor ID esta deshabilitado.
        :RESP:
        -
        */
        sensorMask = (cmd->data)[0];
break;
case DISTANCE_SENSOR_GET_MASK:
        /* Obtiene el estado de habilitacion de cada uno de los sensores de distancia
        conectados al controlador.
        :DATO:
        -
        :RESP:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor a
        habilitar o deshabilitar. Si 2^ID = 1 entonces el sensor ID esta habilitado.
        Si 2^ID = 0 entonces el sensor ID esta deshabilitado.
        */
        // Envio la mascara de sensores
        response.data[0] = sensorMask;
        // Corrijo el largo del paquete
        response.len++;
break;
case DISTANCE_SENSOR_GET_VALUE:
        /* Obtiene el valor promedio de la entrada de los sensores indicados.
        :DATO:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual obtener la lectura.
        :RESP:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual proviene el la lectura de distancia. Secuencia de numeros enteros
        positivos de 16 bits en el rango desde 0x0000 hasta 0x03FF, con el valor de
        la lectura que representa la distancia al objeto. En la secuencia de numeros
        el orden esta dado de izquierda a derecha comenzando por el bit menos
        significativo.
        En el caso del sensor de ultrasonido el rango es desde 0x0000 hasta 0x7594
        que representa la minima y maxima lectura del sensor.
        En el caso del switch, un estado logico bajo se lee como 0x0000 y un estado
        logico alto se lee como 0xFFFF.
        */
        readSensor = (cmd->data)[0];
        requestFrom =   response.to;
        requestCmd = response.cmd;
        sendResponse = false;
        samples = SAMPLES_DEFAULT;
break;
case DISTANCE_SENSOR_GET_ONE_VALUE:
        /* Obtiene el valor de la entrada del sensor indicado. Igual al comando 8.5 pero
        sin realizar un promedio de lecturas.
        :DATO:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual obtener la lectura.
        :RESP:
        Valor de 0x00 a 0x3F donde cada bit representa el ID del sensor
        del cual proviene el la lectura de distancia. Secuencia de numeros enteros
        positivos de 16 bits en el rango desde 0x0000 hasta 0x03FF, con el valor de
        la lectura que representa la distancia al objeto. En la secuencia de numeros
        el orden esta dado de izquierda a derecha comenzando por el bit menos
        significativo.
        En el caso del sensor de ultrasonido el rango es desde 0x0000 hasta 0x7594
        que representa la minima y maxima lectura del sensor.
        En el caso del switch, un estado logico bajo se lee como 0x0000 y un estado
        logico alto se lee como 0xFFFF.
        */
        readSensor = (cmd->data)[0];
        requestFrom =   response.to;
        requestCmd = response.cmd;
        sendResponse = false;
        samples = 1;
break;
case DISTANCE_SENSOR_ALARM_ON_STATE:
        /* Cuando un switch esta presente en el ID: 0x05, establece si se desea o no
```

```
                        recibir una alarma ante cierto cambio de estado en el mismo. Puede ser ante
                        cualquier cambio o sobre un anco ascendente o descendente.
                        :DATO:
                        Valor entre 0x00 y 0x03 con el tipo de cambio ante el cual generar
                        la alarma. Con un 0x00 ignora cualquier cambio en el switch. Se utiliza
                        0x01 para que cualquier cambio en el switch genere el mensaje, 0x02 para
                        que sea solo ante un flanco ascendente y 0x03 para que sea solo ante un
                        flanco descendente.
                        :RESP:
                        -
                        */
#if TRIGGER_TYPE == SWITCH_SENSOR
                        switch ((cmd->data)[0])
                        {
                                case 0x00:
                                        // Ignorar
                                        disable_interrupts(INT_EXT);
                                        break;
                                case 0x01:
                                        // Cualquier cambio
                                        enable_interrupts(INT_EXT);
                                        break;
                                case 0x02:
                                        // flanco ascendente
                                        ext_int_edge(H_TO_L);
                                        enable_interrupts(INT_EXT);
                                        break;
                                case 0x03:
                                        // flanco descendente
                                        ext_int_edge(L_TO_H);
                                        enable_interrupts(INT_EXT);
                                        break;
                                default:
                                        (cmd->data)[0] = 0x00;
                                        break;
                        }
                        alarmType = (cmd->data)[0];
#endif
                break;
                default:
                        response.len++;
                        response.cmd = COMMON_ERROR;
                        response.data[0] = 0x01; // Comando desconocido
                break;
        }

        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

        return;
}
```

B.5. Placa controladora de servo motores

Código fuente de la placa controladora de servo motores.

```
//CCS PCM V4.023 COMPILER

#define CARD_GROUP      SERVO_MOTOR     // Ver protocol.h
#define CARD_ID         0                       // Valor entre 0 y E

// Descripcion de la placa
#define DESC            "SERVOR CONTROL - 1.0" // Maximo DATA_SIZE bytes

/* Modulo Servo - main.c
 * PIC16F88 - MAX232 - SERVO
 *
 *                              PIC16F88
 *               .-----------------------------------.
 *       MOTOR_4 -|RA2/AN2/CVREF/VREF           RA1/AN1|- MOTOR_5
 *           LED -|RA3/AN3/VREF+/C1OUT          RA0/AN0|-
 *           LED -|RA4/AN4/TOCKI/C2OUT    RA7/OSC1/CLKI|- XT CLOCK pin1, 27pF to GND
```

```
 * RST/ICD2:MCLR -|RA5/MCLR/VPP           RA6/OSC2/CLKO|- XT CLOCK pin2, 27pF to GND
 *           GND -|VSS                          VDD|- +5v
 *       MOTOR_1 -|RB0/INT/CCP1       RB7/AN6/PGD/T1OSI|- ICD2:PGD
 *       MOTOR_2 -|RB1/SDI/SDA  RB6/AN5/PGC/T1OSO/T1CKI|- ICD2:PGC
 *  MAX232:R1OUT -|RB2/SDO/RX/DT           RB5/SS/TX/CK|- MAX232:T1IN
 *     ICD2:PGM/ -|RB3/PGM/CCP1             RB4/SCK/SCL|-
 *     MOTOR_3    '-----------------------------------'
 *
 */

#include <16F88.h>
#DEVICE ADC = 10
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#fuses HS,NOWDT,NOPROTECT,NOLVP
#use delay (clock=20000000)

#use rs232(BAUD=115200,PARITY=N,XMIT=PIN_B5,RCV=PIN_B2,BITS=8,ERRORS,TIMEOUT=1,STOP=1,UART1)
#use fast_io(A)
#use fast_io(B)

#byte porta=0x05
#byte portb=0x06

// Led
#bit led1=porta.1
#bit led2=porta.4

// MAX232
#bit tx=portb.5
#bit rx=portb.2

// PWMs
#bit pwm1=portb.0
#bit pwm2=portb.1
#bit pwm3=portb.3
#bit pwm4=porta.2
#bit pwm5=porta.1

#include <../../protocolo/src/protocol.c>
/*
** Variables definidas en protocol.c

short reset; // Variable para hacer el reset
short crcOK; // Informa si el CRC del paquete parseado fue correcto
short sendResponse; // Informa que no debe mandarse la respuesta automatica

char buffer[MAX_BUFFER_SIZE];   // Buffer de recepcion de comandos
int buffer_write;                               // Indice de escritura
int buffer_read;                                // Indice de lectura
int data_length;                                // Largo de los datos en el buffer

struct command_t command;       // Comando parseado
struct command_t response;      // Respuesta

** Implementar las siguientes funciones (usadas por el protocolo)

void init(); // Inicializa puertos y variables
void doCommand(struct command_t * cmd); // Examina y ejecuta el comando

***/

// Software PWM - Minimo 1750 (~0.71ms)
#define PULSE_MIN       1750
// Tiempo maximo que puede durar un pulso - Maximo 6755(~2.71ms)
#define PULSE_MAX       6755
// Tiempo entre pulsos (~25ms -> ~22.29ms fijos de espera)
#define PWM_MAX         62500
// 1 ~ 27.8 cuentas ~ 69.4us
#define DEGREE          27.8f

// Valor que representa el ancho del pulso para cada servo
```

```
long pwm_t[5];
// Angulo de cada servo
long pos[5];
// On/Off de cada servo
short servo[5];

void init()
{
        // Inicializa puertos
        set_tris_a(0b11100001);
        set_tris_b(0b11110100);

        // Seteo el Timer1 como fuente interna
        setup_timer_1(T1_INTERNAL | T1_DIV_BY_2);
        set_timer1(0);

        // Variable para hacer el reset
        reset = false;

        // Valor que representa el ancho del pulso para cada servo
        pwm_t[0] = PULSE_MIN;
        pwm_t[1] = PULSE_MIN;
        pwm_t[2] = PULSE_MIN;
        pwm_t[3] = PULSE_MIN;
        pwm_t[4] = PULSE_MIN;

        // Angulo de cada servo
        pos[0] = 0;
        pos[1] = 0;
        pos[2] = 0;
        pos[3] = 0;
        pos[4] = 0;

        // On/Off de cada servo
        servo[0] = 0;
        servo[1] = 0;
        servo[2] = 0;
        servo[3] = 0;
        servo[4] = 0;

        // Activo los servos segun este o no habilitado
        pwm1 = servo[0];
        pwm2 = servo[1];
        pwm3 = servo[2];
        pwm4 = servo[3];
        pwm5 = servo[4];

        return;
}

void main()
{
        short check_comm = 0;
        long tmr1;

        // Control de servomotores
        init();

        // Init del protocol
        initProtocol();


        // FOREVER
        while(true)
        {
                // Software PWM

                // Tomo el tiempo
                tmr1 = get_timer1();

                // Es hora de reiniciar el pulso?
                if (tmr1 >= PWM_MAX)
                {
                        // Detiene el analisis de comandos
```

```
                        check_comm = 0;

                        // Inicio del periodo
                        set_timer1(0);
                        // Activo los servos segun este o no habilitado
                        pwm1 = servo[0];
                        pwm2 = servo[1];
                        pwm3 = servo[2];
                        pwm4 = servo[3];
                        pwm5 = servo[4];

                } else
                // Llego al final del pulso?
                if (tmr1 >= PULSE_MAX)
                {
                        // Pone las salidas a 0
                        pwm1 = 0;
                        pwm2 = 0;
                        pwm3 = 0;
                        pwm4 = 0;
                        pwm5 = 0;
                        // Analiza si hay comandos para ser atendidos
                        check_comm = 1;

                } else {
                        // Tomo el tiempo
                        tmr1 = get_timer1();

                        // Es tiempo de desactivar el PWM?
                        if (tmr1 >= pwm_t[0])
                                pwm1 = 0;
                        if (tmr1 >= pwm_t[1])
                                pwm2 = 0;
                        if (tmr1 >= pwm_t[2])
                                pwm3 = 0;
                        if (tmr1 >= pwm_t[3])
                                pwm4 = 0;
                        if (tmr1 >= pwm_t[4])
                                pwm5 = 0;
                }

                // Protocolo
                if (check_comm == 1)
                        runProtocol(&command);
        }

        return;
}

/* Verifica que el comando sea valido y lo ejecuta */
void doCommand(struct command_t * cmd)
{
        int crc, i, len;

        // Calculo del CRC
        crc = generate_8bit_crc((char *)cmd, cmd->len, CRC_PATTERN);

        // CRC ok?
        if (cmd->crc != crc)
        {
                // Creo respuesta de error
                response.len = MIN_LENGTH + cmd->len + 2 + 1;
                response.to = cmd->from;
                response.from = THIS_CARD;
                response.cmd = COMMON_ERROR;
                response.data[0] = 0x00;
                // Agrego el paquete que contiene el error de CRC
                response.data[1] = cmd->len;
                response.data[2] = cmd->to;
                response.data[3] = cmd->from;
                response.data[4] = cmd->cmd;
                // Campo data
                len = cmd->len - MIN_LENGTH;
                for (i = 0; i < len; i++)
```

```
                response.data[5 + i] = (cmd->data)[i];
        // CRC erroneo
        response.data[5 + len] = cmd->crc;
        // CRC esperado
        response.data[5 + len + 1] = crc;
        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

        crcOK = false;
        return;
}

crcOK = true;

// Minimo todos setean esto
response.len = MIN_LENGTH;
response.to = cmd->from & 0x77;
response.from = THIS_CARD;
response.cmd = cmd->cmd | 0x80;

switch (cmd->cmd)
{
        // Comandos comunes
        case COMMON_INIT:
                init();
                // Enviar la descripcion de la placa en texto plano
                strcpy(response.data, DESC);
                response.len += strlen(response.data);
        break;
        case COMMON_RESET:
                // Enviar la descripcion de la placa en texto plano
                strcpy(response.data, DESC);
                response.len += strlen(response.data);
                // Reset!
                reset = true;
        break;
        case COMMON_PING:
                // No hace falta hacer mas nada
        break;
        case COMMON_ERROR:
                // Por ahora se ignora el comando
        break;

        /* Comandos especificos */

        case SERVO_MOTOR_SET_POSITION:
                /* Determina la posicion en la que debe colocarse el
                servo motor indicado.
                :DATO:
                Valor de 0x00 a 0x04 que determina el id del servo al
                que se le aplicara la posicion. Valor entre 0x00 y 0xB4
                que representa el rango de 0 a 180 con 1 de presicion.
                :RESP:
                -
                */
                i = ((cmd->data)[0] & 0x07); // Servo destinatario
                if (i < 5)
                {
                        servo[i] = 1;
                        pos[i] = (unsigned char)((cmd->data)[1]);
                        pwm_t[i] = PULSE_MIN + pos[i] * DEGREE;
                }
        break;
        case SERVO_MOTOR_SET_ALL_POSITIONS:
                /* Determina las posiciones en la que deben colocarse
                cada uno de los servomotores
                :DATO:
                Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno
                para cada uno de los servos conectados al controlador.
                Cada valor representa el rango de 0 a 180 con 1 de presicion.
                :RESP:
                -
                */
                for (i = 0; i < 5; i++)
```

```
        {
                servo[i] = 1;
                pos[i] = (unsigned char)((cmd->data)[i]);
                pwm_t[i] = PULSE_MIN + pos[i] * DEGREE;
        }
break;
case SERVO_MOTOR_GET_POSITION:
        /* Obtiene la ultima posicion del servomotor indicado.
        :DATO:
        Valor de 0x00 a 0x04 que determina el id del servo del que
        se requiere la posicion.
        :RESP:
        Valor de 0x00 a 0x04 que determina el id del servo del que
        se requirio la posicion. Valor entre 0x00 y 0xB4 que representa
        el rango de 0 a 180 con 1 de presicion.
        */
        i = ((cmd->data)[0] & 0x07); // Servo destinatario
        if (i < 5)
        {
                response.data[0] = pos[i];
                response.len++;
        }
break;
case SERVO_MOTOR_GET_ALL_POSITIONS:
        /* Obtiene las ltimas posiciones de todos los servomotor
        conectados al controlador.
        :DATO:
        -
        :RESP:
        Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para
        cada uno de los servos conectados al controlador. Cada valor
        representa el rango de 0 a 180 con 1 de presicion.
        */
        response.data[0] = pos[0];
        response.data[1] = pos[1];
        response.data[2] = pos[2];
        response.data[3] = pos[3];
        response.data[4] = pos[4];
        response.len += 5;
break;
case SERVO_MOTOR_SET_SERVO_SPEED:
        /* Determina la velocidad a la que el servomotor indicado
        llegara a la posicion.
        :DATO:
        Valor de 0x00 a 0x04 que determina el id del servo al que
        se le aplicara la velocidad. Valor entre 0x00 y 0xB4,
        velocidad en grados por segundo.
        :RESP:
        -
        */
break;
case SERVO_MOTOR_SET_ALL_SPEEDS:
        /* Determina las velocidades a la que cada uno de los
        servomotores llegara a la posicion indicada.
        :DATO:
        Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno
        para cada uno de los servos conectados al controlador.
        Cada valor representa a la velocidad en grados por segundo.
        :RESP:
        -
        */
break;
case SERVO_MOTOR_GET_SERVO_SPEED:
        /* Obtiene la velocidad asignada al servomotor indicado.
        :DATO:
        Valor de 0x00 a 0x04 que determina el id del servo del que
        se requiere la velocidad.
        :RESP:
        Valor de 0x00 a 0x04 que determina el id del servo del que
        se requirio la velocidad. Valor entre 0x00 y 0xB4, velocidad
        en grados por segundo.
        */
break;
case SERVO_MOTOR_GET_ALL_SPEEDS:
```

```
                        /* Obtiene las velocidades de cada uno de los servomotor
                        conectados al controlador.
                        :DATO:
                        -
                        :RESP:
                        Consta de 5 valores entre 0x00 y 0xB4 concatenados, uno para
                        cada uno de los servos conectados al controlador. Cada valor
                        representa a la velocidad en grados por segundo.
                        */
                break;
                case SERVO_MOTOR_FREE_SERVO:
                        /* Deja de aplicar fuerza sobre el servo indicado.
                        :DATO:
                        Valor de 0x00 a 0x04 que determina el id del servo a liberar.
                        :RESP:
                        -
                        */
                        i = ((cmd->data)[0] & 0x07); // Servo destinatario
                        if ((i < 5) && (i >= 0))
                        {
                                servo[i] = 0;
                        }
                break;
                case SERVO_MOTOR_FREE_ALL_SERVOS:
                        /* Deja de aplicar fuerza sobre cada uno de los servomotor
                        conectados al controlador.
                        :DATO:
                        -
                        :RESP:
                        -
                        */
                        servo[0] = 0;
                        servo[1] = 0;
                        servo[2] = 0;
                        servo[3] = 0;
                        servo[4] = 0;
                break;
                default:
                        response.len++;
                        response.cmd = COMMON_ERROR;
                        response.data[0] = 0x01; // Comando desconocido
                break;
        }

        // CRC de la respuesta
        response.crc = generate_8bit_crc((char *)(&response), response.len, CRC_PATTERN);

        return;
}
```

C. Costo del prototipo

Principales costos de armado del prototipo.

Cantidad	Detalle	Unitario	Total
1	Netbook	2451	2451
4	Placa genérica	97	388
2	Placa controladora motor DC	151	302
4	Placa controladora de sensores	106	424
2	Motorreductor IGNIS MR2-FA	50	100
10	Telémetro infrarrojo GP2D120	35	280
1	Sensor de ultrasonido SRF05	106	106
5	Sensor de piso CNY70	10	50
2	Rueda 100x26	30	60
1	Batería $12v$ $7Ah$	85	85
10	Microcontrolador PIC16F88	23	230
2	Driver L298	21	42
1	Adaptador USB - Serial RS232	90	90
1	Cargador de batería automático	68	68
1	Rueda castor 30C	11	11
1	Rueda castor 40C	14	14

Cuadro 23: Lista de materiales.