Microcyc :

Un sistema de desarrollo basado en el microcontrolador 80592 con preprocesador para control borroso

Autor: Avelino Herrera Morales
Tutor: Juan Albino Méndez Pérez

24 de septiembre de 2002

Departamento de Física Fundamental y Experimental, Electrónica y Sistemas
Centro Superior de Informática
Universidad de La Laguna

D. JUAN ALBINO MÉNDEZ PÉREZ, Doctor en Informática y profesor del Centro Superior de Informática de la Universidad de La Laguna.

CERTIFICA:

Que D. Avelino Herrera Morales ha realizado bajo mi dirección el presente proyecto, con título "MICROCYC : UN SISTEMA DE DESARROLLO BASADO EN EL MICRO-CONTROLADOR 80592 CON PREPROCESADOR PARA CONTROL BORROSO". Con esta fecha, autorizo la presentación del mismo.

La Laguna, Septiembre de 2002.
El Director del Proyecto.

A Alexis por toda la ayuda prestada.

A mis padres.
A mi novia.
A todos los amigos de la facultad.

Índice General

Índice de Tablas

Índice de Figuras

Capítulo 1

Introducción

En el presente proyecto se ha implementado un sistema hardware y software de desarrollo basado en el microcontrolador 80592. Los módulos básicos del sistema desarrollado son el microcontrolador, los módulos de memoria RAM (32 Kbytes), el módulo de memoria ROM (128 Kbytes) y un conversor DA para dotar al sistema de salidas analógicas. Para facilitar el desarrollo y la depuración de aplicaciones sobre el microcontrolador también se ha implementado un emulador de FlashROM a partir de memorias RAM.

Además de la implementación hardware del prototipo, en este proyecto se ha desarrollado una aplicación para implementar controladores borrosos en el sistema de desarrollo. Para ello se creó un traductor de lenguaje, para traducir código de alto nivel (con el que se define el controlador borroso) a lenguaje C. Como aplicación se consideró el control de un prototipo a escala de un puente de grúa.

1.1 Objetivos

A continuación se enumeran los objetivos planteados en este proyecto:

1. Contrucción de un sistema hardware basado en el procesador 80592 de la familia MCS51.
2. Construcción de un emulador ROM para agilizar el desarrollo de software para el prototipo.
3. Implementación de un controlador borroso en el prototipo.
4. Aplicación del control borroso sobre una planta real.

1.1.1 El sistema hardware en torno al 80592

El prototipo desarrollado es un sistema sencillo con una configuración básica formada por el procesador, una memoria FlashROM para código y una memoria RAM estática para datos. Las entradas y salidas de este sistema se corresponden con las entradas y salidas propias del microcontrolador (entradas y salidas digitales y varias entradas analógicas).

El microcontrolador 80592 está basado en la familia de microcontroladores de 8 bits MCS51 de Intel y se compone de un núcleo básico de un 8051 al que se le han añadido 8 entradas analógicas, 2 salidas PWM, un interface de bus CAN2.0A, un timer adicional de propósito general (T2) y un timer de watch dog (T3).

Este sistema hardware construido alrededor del 80592 posee 32 Kbytes de memoria RAM de datos y 64 KBytes de memoria Flash de programa. El prototipo se ha completado con la inclusión de un conversor digital-analógico que permite disponer de una salida analógica. Ésto resulta especialmente útil para la implementación de lazos de control.

1.1.2 El emulador ROM

Para facilitar el desarrollo y la depuración de aplicaciones para el 80592, se ha añadido un emulador de FlashROM. Dicho dispositivo emula el funcionamiento de una memoria flash 29F010, tratándose, en realidad, de una memoria RAM alimentada de forma independiente y escribible desde un PC. El emulador permite emular una ROM de 32 Kbytes al estar construido alrededor de una RAM estática 62256.

Un emulador ROM permite aliviar la carga física de desarrollo al poder estar el prototipo permanentemente en contacto con un PC desde el que se vuelca el código para ser ejecutado. De esta forma evitamos el uso de memorias FlashROM para el proceso de depuración, utilizando pastillas de memoria no volátil sólo para grabar programas para su explotación.

1.1.3 El controlador borroso

Después de la implementación hardware de los dos puntos anteriores en el proyecto se planteó el desarrollo de una solución software para control borroso. Dicho control borroso se aplicó a un prototipo de grúa (control de posición y control de las oscilaciones de la carga colgante).

El objetivo de la implementación es el control de posición del prototipo de grúa simultaneado con un control de las oscilaciones de la carga, procurando que ésta se mantenga lo más vertical posible a la vez que se posiciona el vehículo en el punto deseado.

1.2 Motivaciones

Existen múltiples sistemas de desarrollo en el mercado basados en microcontroladores de la familia MCS51. En este caso se ha optado por la fabricación desde cero, de un sistema de desarrollo utilizando placas de prototipos. Al interés propio de desarrollar un sistema de control borroso se le añade el hecho de implementarlo en un sistema pequeño de 8 bits, con 64 Kbytes de memoria de programa y 32 Kbytes de memoria de datos.

Capítulo 2

El microcontrolador

En este capítulo describiremos el funcionamiento y las características del microcontrolador 80592: Repertorio de instrucciones, organización de la memoria, modos de direccionamiento que soporta y otras características adicionales tales como el conversor analógico-digital que posee, las salidas PWM, los timers, las fuentes de interrupción y el controlador CAN integrado en el chip, compatible con la revisión 2.0A del protocolo.

2.1 El 80592

A principios de los años 80, Intel introduce el microcontrolador 8051 (perteneciente a la familia MCS51). Se trata de un microcontrolador fabricado, en su versión original, con tecnología HMOS. Tanto el 8051, como otros microcontroladores (8048, 8031, 8052, etc) forman parte de la llamada *familia MCS51*. Dicha familia de procesadores está construida alrededor de un núcleo básico (denominado *núcleo 8051*) con funcionalidades adicionales añadidas según la versión.

La familia MCS-51 se encuentra muy extendida en el campo industrial, habiendo muchos autómatas y tarjetas chip construidos con ellos (por citar algunos ejemplos). Por otro lado Intel firmó acuerdos de producción con otras empresas tales como Oki, Philips, Siemens o Signetics para que éstas pudiesen fabricar microcontroladores nuevos utilizando el núcleo básico del 8051 (añadiéndole DACs, ADCs, controladores de buses tales como el I2C, el CAN, etc)

En nuestro caso se ha utilizado el microcontrolador 80592 de Philips Semiconductors. Se trata, pues, de un procesador basado en el núcleo 8051 de Intel (*totalmente* compatible a nivel software con su antecesor de Intel) al que se le ha añadido una interfaz CAN, 8 entradas analógicas multiplexadas y un ADC de 10 bits, 6 puertos E/S de 8 bits, 15 fuentes de interrupción con 2 niveles de prioridad y con una frecuencia de reloj de entre 1.8 y 16 MHz. Cada uno de los bits de los puertos posee una función alternativa a parte de la que tiene asignada por defecto (ver figura 2.1). En la figura 2.2 podemos ver el

encapsulado del microcontrolador utilizado en este proyecto.

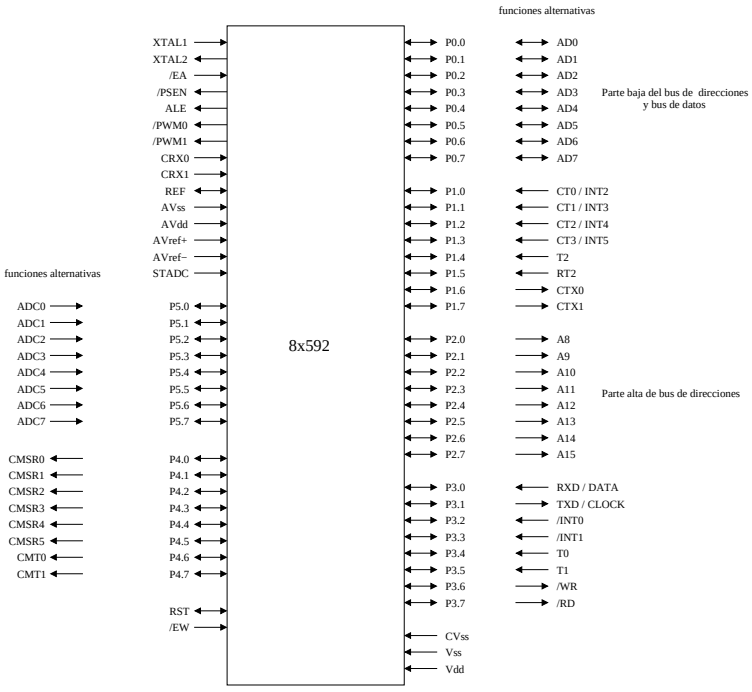


Figura 2.1: Pines del 80592

El 80592, al igual que todos los sucesores del 8051, es un microcontrolador de 8 bits, con una ALU de 8 bits que permite realizar algunas operaciones de 16 bits (multiplicación y división) y algunas ooperaciones de 1 bit (mediante direccionamiento de bit, operaciones lógicas). Existe separación lógica entre los datos y las instrucciones aun siendo el bus de datos y de direcciones común a ambos mundos. Posee múltiples modos de direccionamiento:

- Inmediato.
- Directo.
- Por registro.
- Indirecto por registro.
- Indirecto indexado por registro.
- De bit.

A lo largo de este capítulo se irá profundizando en cada una de las características de este microcontrolador.

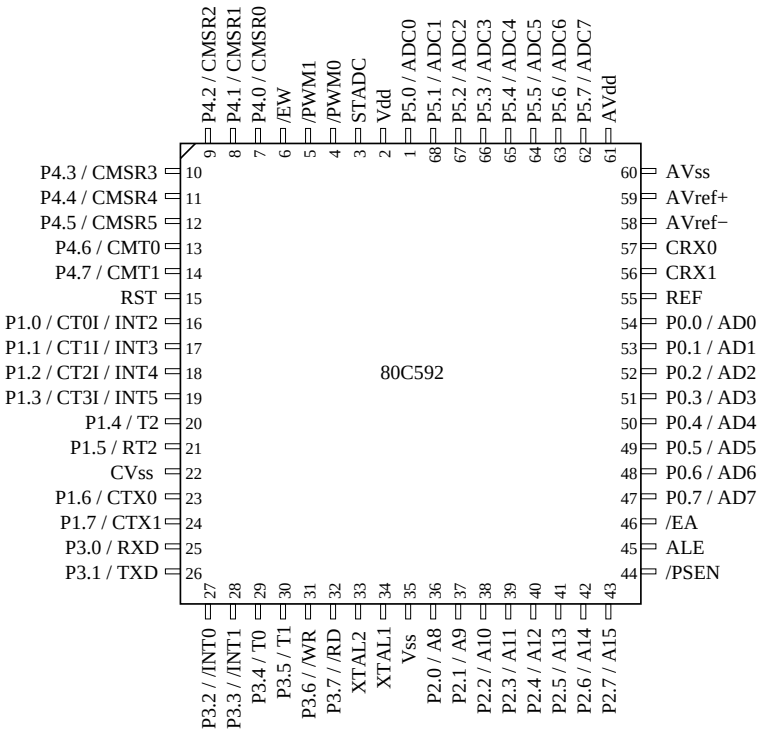


Figura 2.2: Encapsulado del 80592

2.2 Registros

En el 80592, al igual que en el resto de congéneres de la familia MCS51, los registros del procesador se encuentran mapeados en la memoria RAM interna. Se pueden diferenciar entre 2 tipos de registros:

- Registros de función especial.
- Registro de banco.

Dentro de los 256 bytes de memoria RAM principal interna del 80592, los registros de función especial se encuentran en los 128 últimos bytes cuando se acceden mediante direccionamiento directo. Son estos SFRs los que permiten el acceso a puertos, al acumulador, al registro de multiplicación, etc. A medida que se vayan explicando las capacidades propias del microcontrolador se irá profundizando en el resto de SFRs que posee.

Por otro lado tenemos los registros de banco. El 80592 posee 4 bancos de registros, cada banco con 8 registros: R0 a R7, y alojados en la parte baja de la RAM pricniapl interna. Sólo un banco puede estar activo en un instante determinado. Estos registros de banco son de utilidad general y sólo los R0 y R1 de cada banco tienen como utilidad especial el poder servir como punteros en algunos modos de direccionamiento.

2.3 Organización de la memoria

El microcontrolador permite direccionar 64 Kbytes de memoria de programa externa, posee 256 bytes de memoria RAM interna de los cuales los 128 primeros son direccionables de forma directa e indirecta, mientras que los 128 restantes sólo son direccionables indirectamente (ver figura 2.3). Existe, además, una zona de memoria superpuesta a esta zona de acceso indirecto y que contiene los SFR (*Special Function Registers*). Cuando realizamos un acceso indirecto accedemos a la RAM interna indirecta y cuando hacemos un acceso directo accedemos a los SFR. Este procesador posee un tipo de direccionamiento adicional a los tradicionales, que es el direccionamiento de bit. Dicho direccionamiento sólo es aplicable a algunas zonas de los SFR (el puerto P0, por ejemplo está representado en la RAM interna por un SFR situado en la dirección 80h y, además sus bits P0.0 a P0.7 pueden ser accedidos de forma individual mediante direccionamiento de bit en las direcciones 80h a 87h).

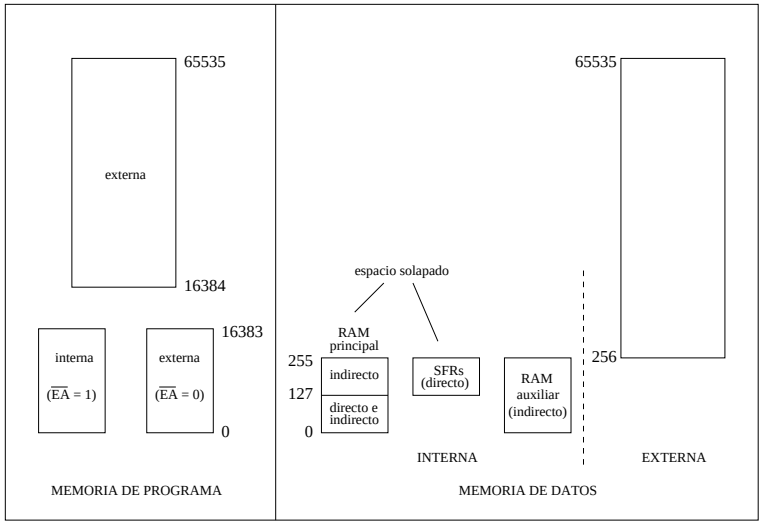


Figura 2.3: Mapa de memoria del 8x592

Si la RAM interna no es suficiente para nuestra aplicación, es posible dotar al sistema de una RAM externa. El único inconveniente de esta RAM externa es que, aparte de ser más lenta por razones obvias, sólo es posible acceder a ella mediante la instrucción MOVX. No es posible realizar, por tanto operaciones directamente sobre variables alojadas en esta RAM.

La RAM auxiliar aunque es interna, no se trata como tal, ya que es una zona de 256 bytes de RAM direccionables como si de una RAM externa se tratase, esto es: al direccionar un dato en la memoria RAM externa entre la dirección 0 y la 255 accederemos realmente a la RAM auxiliar (alojada físicamente dentro del microcontrolador), mientras que si direccionamos datos de la RAM externa entre la dirección 256 y la 65535 accederemos a la RAM externa y físicamente externa.

Toda la familia de microcontroladores MCS51 posee un bus de datos y de direcciones al estilo 8086, esto es, para ahorrar pines, los datos y las direcciones salen al exterior en

momentos del ciclo del bus diferentes. El puerto 2 (bits P2.0 a P2.7) saca al exterior la parte alta del bus de direcciones, mientras que el puerto 0 (bits P0.0 a P0.7) en un primer momento del ciclo del bus indica la parte baja del bus de direcciones (con el pin ALE (*address latch enable*) a 1) y en un segundo momento del ciclo emite/lee los 8 bits del bus de datos (con el pin ALE a 0). Es preciso, por tanto, cuando se va a escribir o a leer un dato o una instrucción a través del bus, "recordar"el valor que tenían los 8 bits más bajos de la dirección.

Mediante un latch montado directamente a la salida del puerto P0 podemos almacenar en cada ciclo de bus el valor de la parte baja del bus de direcciones simplemente con conectar la salida ALE del microcontrolador a la entrada LE (*Latch Enable*) del latch.

2.3.1 Memoria de programa

Algunos modelos de 80592 (en concreto los modelos 83592) poseen 16 Kbytes de memoria ROM de programa dentro del propio chip. El modelo que se ha utilizado es el modelo 80592 que no posee esta memoria. Esto significa que el chip depende, para su funcionamiento de una memoria externa de programa.

El pin etiquetado como /EA (ver figura 2.1) permite indicar al microcontrolador de donde debe leer los primeros 16 Kbytes de memoria de programa (/EA = 0, para utilizar una pastilla externa, y /EA = 1 para utilizar la ROM interna; ver la figura 2.3). En nuestro caso se ha configurado el pin /EA permanentemente a 0, ya que se trata de un modelo sin ROM interna y, desde el primer byte, el código debe ser leido desde el exterior del microcontrolador.

El vector de reset se sitúa en la dirección 0x0000 y los vectores de interrupción se sitúan en las direcciones 0x0003, 0x000B, 0x0013, 0x001B... y así sucesivamente hasta completar los 15 vectores de interrupción soportados por el microcontrolador. La práctica habitual consiste en colocar en la dirección 0x0000 una instrucción "`ljmp funcion_main`"para así dejar libres las entradas de los vectores de interrupción. De esta manera, al ser el vector 0x0000 el de reset, cuando se reinicie el procesador se ejecutará esta instrucción "`ljmp`"que hará una salto hasta el programa principal a ejecutar (si colocásemos directamente el código principal a partir de la dirección 0x0000 no podríamos utilizar los vectores de interrupción que comienzan casi inmediatemente, a partir de la dirección 0x0003).

2.3.2 Memoria de datos

La memoria de datos del 80592 está organizada según se muestra en la tabla 2.1.

La RAM principal puede ser accedida tanto de forma directa como indirecta, la RAM auxiliar sólo puede ser accedida mediante la instrucción MOVX, como si fuese RAM externa. A la RAM externa se accederá cuando se intente direccionar con la instrucción

Memoria	Tipo	Tamaño	Localización	Direccionamiento		Punteros
				Directo	Indirecto	
Interna	Principal	256 bytes	0 a 127	X	X	R0 y R1
			128 a 255	–	X	
	Auxiliar	256 bytes	0 a 255	–	X	R0, R1 y DPTR
	SFRs	128 bytes	128 a 255	X	–	–
Externa	–	65280 bytes	256 a 65535	–	X	R0,R1 y DPTR

Tabla 2.1: Organización de la memoria de datos del 80592

MOVX más allá de los 256 bytes de esta RAM auxiliar. En el 8051 original no existe RAM auxiliar y la RAM externa es accedida desde la dirección 0x0000.

La RAM interna

La RAM interna está dividida en 3 bloques:

- La RAM principal de 256 bytes. Los 128 primeros direccionables directa e indirectamente, y los 128 últimos direccionables sólo de forma indirecta.
- Los SFR o registros de función especial a los que se accede mediante direccionamiento directo de los 128 últimos bytes de la RAM principal.
- La RAM auxiliar a la que sólo se puede acceder mediante la instrucción MOVX y que hace las veces de RAM externa dentro del chip, de 256 bytes de tamaño.

En los primeros 128 bytes de la RAM interna principal (direccionables tanto de forma directa como indirecta) nos encontramos con varias zonas:

- Entre la dirección 0x00 y la 0x1F hay 4 bancos de registros, con 8 registros por banco (el banco 0 entre 0x00 y 0x07, el banco 1 entre 0x08 y 0x0F, el banco 2 entre 0x10 y 0x17, y el banco 3 entre 0x18 y 0x1F). Cada uno de los registros de un banco se denominan R0 a R7 y sólo puede haber un banco activo en cada instante.
- Entre la dirección 0x20 y la 0x2F hay una zona de 128 bits direccionable a nivel de bit (direcciones de bit entre 0x00 y 0x7F). Para uso general.
- Entre la dirección 0x30 y la 0x7F, 80 bytes para uso general.

En la figura 2.4 podemos ver un gráfico que muestra la organización de la RAM interna principal del microcontrolador.

Entre la dirección 0x80 y la 0xFF y direccionables de forma directa, tenemos los SFR o Special Function Registers. Se trata de una serie de registros especiales que poseen funciones específicas. Entre los más importantes destacan:

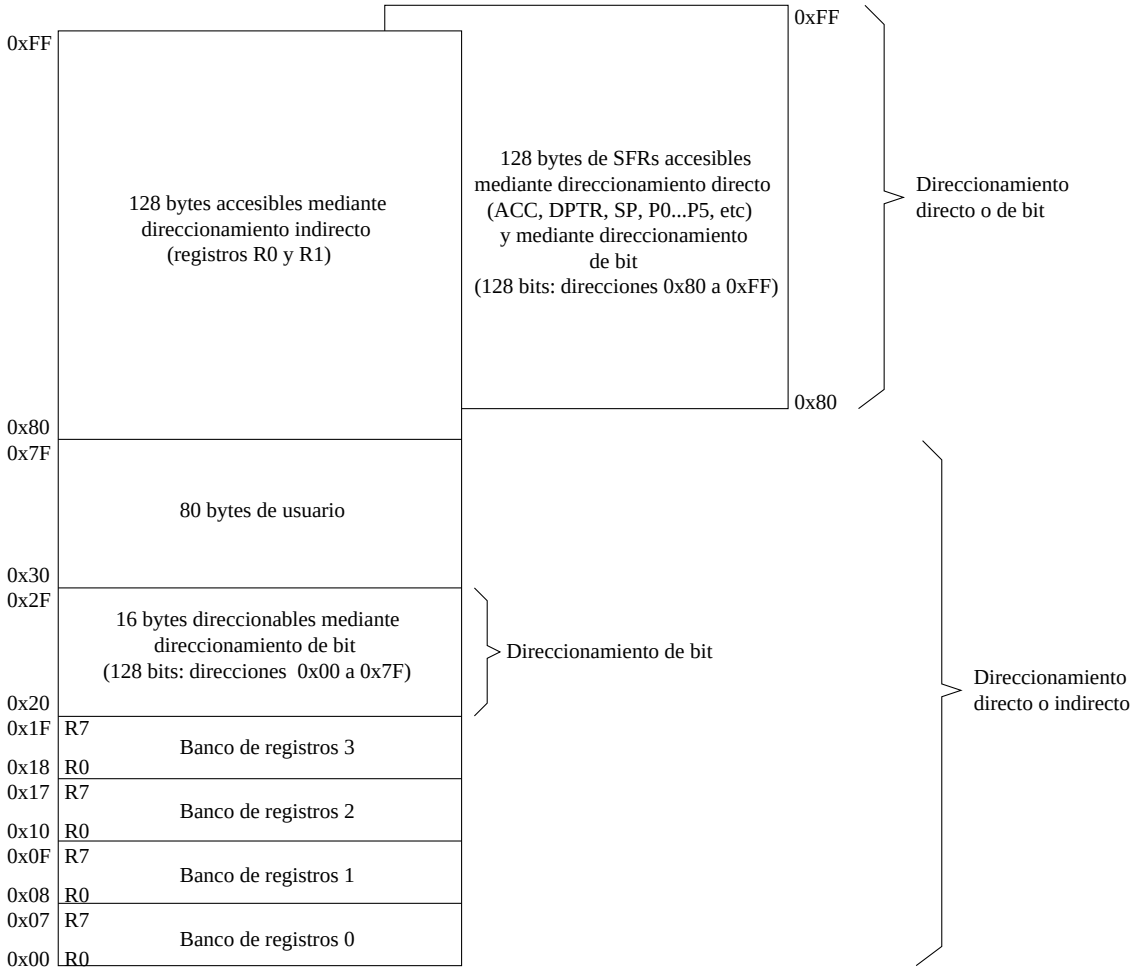


Figura 2.4: Organización de la RAM principal del 80592

SP en 0x81: El puntero de pila.

DPL en 0x82: Parte baja del registro puntero DPTR.

DPH en 0x83: Parte alta del registro puntero DPTR.

PSW en 0xD0: La palabra de estado del procesador. Posee los principales flags del procesador direccionables a nivel de bit:

Bit 7 (CY) en 0xD7: Bit de acarreo.

Bit 6 (AC) en 0xD6: Acarreo auxiliar para BCD.

Bit 5 (F0) en 0xD5: Bit de usuario.

Bit 4 (RS1) en 0xD4: Bit 1 de selección de banco.

Bit 3 (RS0) en 0xD3: Bit 0 de selección de banco.

Bit 2 (OV) en 0xD2: Indicador de desbordamiento.

Bit 0 (P) en 0xD0: Indicador de paridad.

Mediante los bits RS0 y RS1 seleccionamos que banco de registros (0 a 3) deseamos utilizar en cada instante.

ACC en 0xE0: El acumulador para operaciones aritméticas.

B en 0xF0: Registro para multiplicación y división.

Hay que señalar que, aunque el registro del acumulador es el ACC, en el código ensamblador se utiliza el mnemónico "A"para hacer referencia al acumulador. Para el resto de registros de función especial el nombre del registro coincide con su denominación en el código ensamblador.

Tenemos además un SFR por cada puerto del microcontrolador, para el control de los timers, para el control de las comunicaciones CAN, para el control de la UART, para la lectura de las entradas analógicas, y así con todas las peculiaridades propias del chip.

El 80592 basa su compatibilidad con el 8051 original en el hecho de que los registros heredados de este último están alojados en las mismas direcciones, y los nuevos SFRs para las nuevas capacidades del 80592 simplemente se han colocado en los huecos dejados por Intel en esta zona de memoria.

El listado detallado de los registros de función especial del 80592 así como su funcionamiento y modo de empleo se pueden consultar con detalle en la hoja de datos del microcontrolador.

2.4 Puertos de entrada/salida

El 80592 posee 6 puertos de entrada/salida, etiquetados como P0, P1, P2, P3, P4 y P5. Los cuatro primeros puertos (P0 al P3) son los mismos del 8051 original. Cada uno de los 6 puertos se puede utilizar como puerto estándar bidireccional y también como puerto especial para determinados propósitos (acceder a una memoria de código y/o de datos externa, acceder a una línea serial, etc). Ver la tabla 2.2.

2.5 Modos de direccionamiento

El 80592 posee un núcleo 8051 y, por tanto, todos los modos de direccionamiento son los mismos que los de su antecesor.

2.5.1 Direccionamiento inmediato

En el propio código de instrucción es donde se encuentra el operando (memoria de programa).

Puerto	Tipo	Función	Notas
0	E/S	Parte baja del bus de direcciones y bus de datos para memorias externas	
1	E/S	Entradas de captura del timer 2 y entradas de interrupción INT2 a INT5	
		Entrada evento del timer 2	Desde un contador externo
		Entrada de reset del timer 2	De un circuito de reset externo
		Salida 0 del controlador CAN	
		Salida 1 del controlador CAN	
2	E/S	Parte alta del bus de direcciones	
3	E/S	Entrada serial a la UART	
		Salida serial de la UART	
		Petición interrupción 0	
		Petición interrupción 1	
		Entrada timer 0	Entradas para conteo
		Entrada timer 1	
		Señal /WR para la memoria externa	
		Señal /RD para la memoria externa	
4	E/S	Comparación y set/reset	Salidas de comparación de timer 2
5	E	8 canales de entrada del ADC	

Tabla 2.2: Funciones alternativas para cada uno de los puertos del 80592

```
mov    b,#47          ; carga en el SFR b el valor 47
mov    a,#CONSTANTE   ; carga en el acumulador el valor CONSTANTE
```

2.5.2 Direccionamiento directo

Lo que se referencia en este caso es una dirección directa dentro de la RAM principal del 80592.

```
mov    a,0x83  ; carga en el acumulador el valor contenido en la dir 0x83
```

2.5.3 Direccionamiento por registro

Los operandos son registros del banco de registros activo.

```
mov    a,r0    ; carga en el acumulador el contenido del registro r0
mov    b,r3    ; carga en el SFR b el contenido del registro r3
```

2.5.4 Direccionamiento indirecto por registro

En este tipo de direccionamiento se utiliza el contenido de los registros R0 y R1, como puntero para cargar o guardar datos dentro de la RAM interna.

```
mov    r0,#0x40    ; carga el valor 0x40 en R0
mov    a,@r0       ; carga en el acumulador el contenido del byte 0x40
```

2.5.5 Direccionamiento indirecto indexado por registro

Mediante este modo de direccionamiento sólo podemos acceder en modo lectura a la memoria de programa y la única instrucción que permite realizarlo es la instrucción `MOVC A,@A+reg_base`, siendo `reg_base` DPTR (el puntero de uso general de 16 bits) o PC (el contador de programa).

```
inc    a          ; incrementamos a para compensar la instruccion ret
movc   a,@a+pc    ; a es cargado con un 0x10 si a valia 0 antes del inc
ret
db     0x10
db     0x18
db     0x49
```

2.5.6 Direccionamiento de bit

Permite direccionar algunas zonas de la memoria RAM principal así como algunas zonas de los SFR a nivel de bit.

```
mov    c,p2.7     ; escribe el bit 7 del puerto 2 en el bit CY
orl    c,#0x02    ; CY := CY OR (bit de la direccion 0x02)
anl    c,/p4.1    ; CY := CY AND (inversa del bit 1 del puerto 4)
mov    0x33,c     ; Escribe en la direccion de bit 0x33 el bit CY
```

2.6 Repertorio de instrucciones

El repertorio de instrucciones del 80592 es de 256 instrucciones (el mismo repertorio que su padre, el 8051). Tenemos 5 grupos de instrucciones.

2.6.1 Instrucciones de transferencia

Son las instrucciones que utilizan los mnemónicos `MOV`, `PUSH`, `POP`, `XCH`, `XCHD`, `MOVX` y `MOVC`. En la tabla 2.3 se encuentran relacionados mnemónicos de instrucciones de movimiento de datos que afectan a operandos de la memoria RAM interna (no la auxiliar).

Las instrucciones `PUSH` y `POP` empujan o sacan, respectivamente, de la pila un operando de tipo byte. Es preciso señalar que el puntero de pila es un registro de función especial (SFR) alojado en la dirección 0x81 y que, tras el reset, se inicializa a 0x07 con lo que se coloca inicialmente justo por encima del banco de registros 0; es posible, sin embargo situar la pila del sistema donde queramos con tan solo escribir el valor deseado en este registro. SP se incrementa antes de poner el dato (al hacer un PUSH) y se decrementa después de sacar el dato (POP).

En la tabla 2.4 tenemos las instrucciones de movimiento de datos que afectan a la RAM auxiliar y a la RAM externa al microcontrolador.

Además de las instrucciones arriba citadas, existen un par de instrucciones más que permiten recoger datos de la memoria de programa. Las instrucciones `MOVC A,@A+DPTR` y `MOVC A,@A+PC`, cargan en el acumulador el valor almacenado en la dirección A+DPTR o A+PC de la memoria de programa.

Mnemónico	**Direccionamiento**				**Notas**
	Dir	Ind	Reg	Inm	
`MOV A,<orig>`	x	x	x	x	
`MOV <dest>,A`	x	x	x		
`MOV <dest>,<orig>`	x	x	x	x	
`MOV DPTR,#dato16`				x	
`PUSH <orig>`	x				
`POP <dest>`	x				
`XCH A,<byte>`	x	x	x		Intercambia los valores
`XCHD A,@Ri`		x			Intercambia el nibble bajo

Tabla 2.3: Instrucciones de transferencia en la RAM interna

Mnemónico	**Notas**
`MOVX A,@Ri`	A := valor apuntado por Ri
`MOVX @Ri,A`	Posición apuntada por Ri := A
`MOVX A,@DPTR`	A := valor apuntado por DPTR
`MOVX @DPTR,A`	Posición apuntada por DPTR := A

Tabla 2.4: Instrucciones de transferencia en la RAM externa y auxiliar

2.6.2 Instrucciones aritméticas

Las instrucciones aritméticas están agrupadas en los mnemónicos `ADD`, `ADDC`, `SUBB`, `INC`, `DEC`, `MUL`, `DIV` y `DA`. En la tabla 2.5 se puede observar el uso de estos mnemónicos.

Nótese el hecho de que existen dos instrucciones de suma y sólo una de resta. La instrucción `ADD` realiza una suma sin tener en cuenta el acarreo, mientras que la instrucción `ADDC` realiza la suma teniendo en cuenta el indicador de acarreo (para realizar sumas consecutivas o de precisión mayor que 8 bits). Para restar existe una única instrucción `SUBB` que siempre tiene en cuenta el acarreo; esto es, cuando queramos hacer una resta sin tener en cuenta el acarreo deberemos borrar el indicador de acarreo a mano por si estuviese activado de anteriores operaciones.

Mnemónico	**Direccionamiento**				**Notas**
	Dir	Ind	Reg	Inm	
`ADD A,<byte>`	x	x	x	x	
`ADDC A,<byte>`	x	x	x	x	
`SUBB A,<byte>`	x	x	x	x	
`INC A`					
`INC <byte>`	x	x	x		
`INC DPTR`					
`DEC A`					
`DEC <byte>`	x	x	x		
`MUL AB`					$B,A_{16} := A \cdot B$
`DIV AB`					$A := A \,/\, B$, $B := A \bmod B$
`DA A`					Ajuste decimal sobre BCD

Tabla 2.5: Instrucciones aritméticas

2.6.3 Instrucciones lógicas

En la tabla 2.6 se enumeran las instrucciones lógicas que posee el 80592.

2.6.4 Instrucciones booleanas

En la tabla 2.7 están enumeradas todas las instrucciones booleanas del 80592 entre las que se incluyen algunos saltos condicionales. Destacar de entre todas las instrucciones la `JBC bit,rel`, que permite relizar un salto condicional a la dirección `rel` si el bit `bit` se encuentra a 1. Si se encuentra a uno, procede a realizar el salto y, además, pone a 0 el bit que acaba de testear. Todo en una sola instrucción de 2 ciclos.

Mnemónico	Direccionamiento				Notas
	Dir	Ind	Reg	Inm	
ANL A,<byte>	x	x	x	x	A := A and valor
ANL <byte>,A	x				
ANL <byte>,#dato	x				
ORL A,<byte>	x	x	x	x	A := A or valor
ORL <byte>,A	x				
ORL <byte>,#dato	x				
XRL A,<byte>	x	x	x	x	A := A xor valor
XRL <byte>,A	x				
XRL <byte>,#dato	x				
CLR A					A := 0
CPL A					A := not A
RL A					Rotación a la izquierda de A
RLC A					Como RL pero con acarreo (CY)
RR A					Rotación a la derecha de A
RRC A					Como RR pero con ararreo (CY)
SWAP A					Intercambia nibbles alto y bajo

Tabla 2.6: Instrucciones lógicas

2.6.5 Instrucciones de ruptura de secuencia

Dentro de este grupo entran las instrucciones de salto tanto condicional como incondicional. Dentro de las instrucciones de salto incondicional (ver la tabla 2.8) destacar la instrucción `JMP @A+DPTR` con la que podemos realizar saltos a direcciones cuyo valor puede ser calculado en tiempo de ejecución.

En la tabla 2.9 se encuentran indicadas las instrucciones de salto condicional.

Casi todas las instrucciones del 80592 modifican de alguna u otra forma los bits CY, OV y AC de la palabra de estado. Dicha palabra de estado, identificada como PSW, es un registro de función especial alojado en la dirección 0xD0. En la tabla 2.10 se puede ver la relación entre cada una de las instrucciones modificadoras de banderas y los valores que toma cada una de las banderas CY, OV y AC como consecuencia de la ejecución de cada instrucción.

2.7 Las salidas PWM

El microcontrolador 80592 posee dos salidas de pulsos modulados en anchura: /PWM0 y /PWM1. Ambas salidas están basadas en un único contador de 8 bits con un *prescaler*. Dicho prescaler es el SFR PWMP, alojado en la dirección 0xFE. La frecuencia de pulsos PWM viene dada por la siguiente ecuación:

Mnemónico	Notas
`ANL C,bit`	CY := CY and bit
`ANL C,/bit`	CY := CY and not bit
`ORL C,bit`	CY := CY or bit
`ORL C,/bit`	CY := CY or not bit
`MOV C,bit`	CY := bit
`MOV bit,C`	bit := CY
`CLR C`	CY := 0
`CLR bit`	bit := 0
`SETB C`	CY := 1
`SETB bit`	bit := 1
`CPL C`	CY := not CY
`CPL bit`	bit := not bit
`JC rel`	Salto si CY=1
`JNC rel`	Salto si CY=0
`JB bit,rel`	Salto si bit=1
`JNB bit,rel`	Salto si bit=0
`JBC bit,rel`	Salto si bit=1 y luego bit := 0

Tabla 2.7: Instrucciones booleanas

$$f_{PWM} = \frac{f_{CLK}}{2 \cdot (PWMP + 1) \cdot 255}$$

siendo PWMP el contenido del SFR del mismo nombre y f_{CLK} la frecuencia del reloj del 80592. La anchura de pulso de cada una de las salidas viene dada por los SFR PWM0 (en la dirección 0xFC, para la salida /PWM0) y PWM1 (en la dirección 0xFD, para la salida /PWM1). En ambos casos:

$$c = \frac{PWMn}{255 - PWMn}$$

Siendo PWMn el valor del SFR correspondiente y c el ciclo de trabajo (*duty cycle*) de la señal de salida. Un valor $c = 0$ indica que la línea PWMn permanece a 0 constantemente, un valor $c = 1$ indica que la línea PWMn correspondiente permanece a 1 constantemente, mientras que valores $0 < c < 1$ indican ratios de tiempo entre el estado 0 y el estado 1 (por ejemplo, con $c = 0.5$ tenemos los semiciclos 1 de la misma anchura que los semiciclos 0).

Mnemónico	Notas
`JMP dir`	Salto corto
`LJMP dir`	Salto largo, 16 bits
`JMP @A+DPTR`	
`ACALL dir`	Llamada corta
`LCALL dir`	Llamada larga, 16 bits
`RET`	
`RETI`	
`NOP`	

Tabla 2.8: Instrucciones de salto incondicional

Mnemónico	Notas
`JZ rel`	Salto si A = 0
`JNZ rel`	Salto si A $\neq$ 0
`DJNZ <byte>,rel`	Decremento y salto si $\neq$ 0
`CJNE A,<byte>,rel`	Salto si A $\neq$ `<byte>`
`CJNE <byte>,#dato,rel`	Salto si `<byte>` $\neq$ dato

Tabla 2.9: Instrucciones de salto condicional

2.8 El ADC

El 80592 posee un conversor analógico-digital de 10 bits de resolución y con 8 entradas analógicas multiplexadas (correspondientes a los 8 pines del puerto P5). Los SFR asociados al ADC son el ADCON (en la dirección 0xC5) y el ADCH (en la dirección 0xC6).

7	6	5	4	3	2	1	0
ADC.1	ADC.0	ADEX	ADCI	ADCS	AADR2	AADR1	AADR0

Registro ADCON (dirección 0xC5)

7	6	5	4	3	2	1	0
ADC.9	ADC.8	ADC.7	ADC.6	ADC.5	ADC.4	ADC.3	ADC.2

Registro ADCH (dirección 0xC6)

Para iniciar una conversión seleccionamos en los 3 bits menos significativos del registro ADCON la entrada analógica que vamos a leer y ponemos a 1 el bit 3 de ADCON (o, con el bit ADEX a 1, disparamos la conversión desde el exterior mediante un flanco de subida en el pin STDAC). Una vez hecho esto tenemos dos opciones:

1. Hacer polling sobre el bit ADCI hasta que se ponga a 1 para poder leer el valor convertido.

Instrucción	Banderas del PSW			Instrucción	Banderas del PSW		
	CY (bit 7)	OV (bit 2)	AC (bit 6)		CY (bit 7)	OV (bit 2)	AC (bit 6)
ADD	x	x	x	SETB C	1		
ADDC	x	x	x	CLR C	0		
SUBB	x	x	x	CPL C	x		
MUL	0	x		ANL C,bit	x		
DIV	0	x		ANL C,/bit	x		
DA	x			ORL C,bit	x		
RRC	x			ORL C,/bit	x		
RLC	x			MOV C,bit	x		
CJNE	x						

Tabla 2.10: Instrucciones que modifican los indicadores o banderas

2. Configurar las interrupciones para que sea el propio conversor el que nos avise (ver la sección sobre las interrupciones) de que se ha terminado la conversión.

En cualquiera de los casos, una vez finalizada la conversión, el SFR ADCH contendrá los 8 bits más significativos del valor, mientras que los bits menos significativos se encuentran en los bit 6 y 7 de ADCON.

Al final de la conversión es preciso poner a cero el bit 4 (ADCI) por software.

2.9 Timers

El 80592 posee 3 timers de propósito general más un timer adicional de watch dog. De estos 3 timers, los dos primeros, el 0 y el 1, corresponden a los dos timers de su antecesor, el 8051.

2.9.1 Los timers 0 y 1

Los dos primeros timers, el 0 y el 1, provienen del núcleo funcional del 8051. Son timers de 16 bits cada uno y sus valores pueden ser accedidos a través de 4 SFRs:

- Para el timer 0: TL0 (parte baja) en 0x8A y TH0 (parte alta) en 0x8C.
- Para el timer 1: TL1 (parte baja) en 0x8B y TH1 (parte alta) en 0x8D.

Los timers se pueden itilizar para dos labores diferentes:

- Temporización: El timer es incrementado por la propia circuitería de reloj del microcontrolador.
- Conteo: El timer es incrementado por una señal de reloj externa (P3.4 para el timer 0, y P3.5 para el timer 1).

Mediante el registro TMOD (SFR alojado en la dirección 0x89) podemos configurar el funcionamiento de estos dos timers. El registro de 8 bits está dividido en dos: los 4 bits menos significativos controlan el timer 0 y los 4 bit más significativos controlan el timer 1 (ver tabla 2.11).

7	6	5	4	3	2	1	0
GATE	C/T	M_1	M_0	GATE	C/T	M_1	M_0
Timer 1				Timer 0			

Tabla 2.11: El registro de control TMOD (0x89)

Si GATE = 1, el timer correspondiente (x) está habilitado sólo cuando su patilla de entrada INTx está a 1 y el bit TRx del registro TCON está a 1. Si GATE = 0, el timer x estará habilitado si TRx = 1. El bit C/T de cada timer permite seleccionar el modo entre conteo y temporización: si este bit está a 0, el timer actúa como temporizador, mientras que si está a 1, actúa como contador. En modo contador, el timer x se incrementará cada vez que haya una transición de 1 a 0 en la patilla Tx correspondiente (la lectura de las patillas Tx se realiza de forma síncrona en cada período de máquina).

Los bits M_0 y M_1 del registro TMOD permiten seleccionar el modo de funcionamiento de los timers.

- `Modo 0 (`M_1`=0 y `M_0`=0)`. Contador de 13 bits compatible con los de la familia MCS-48.
- `Modo 1 (`M_1`=0 y `M_0`=1)`. Contador de 16 bits.
- `Modo 2 (`M_1`=1 y `M_0`=0)`. Contador de 8 bits con recarga automática. El contador en sí es el TLx (8 bits) y, cuando éste llega a 0, se recarga con el contenido de THx.
- `Modo 3 (`M_1`=1 y `M_0`=1)`.
  - *Timer 0.* El registro TL0 acúa como contador de 8 bits controlado por los bits de control del timer 0. TH0 es configurado como temporizador de 8 bits controlado por los bits del timer 1.
  - *Timer 1.* Este timer permanece detenico.

Asociado a los timers 0 y 1 se encuentra también el registro TCON (0x88), cuyos bits son direccionables individualmente mediante direccionamiento de bit en las direcciones 0x88 a 0x8F. Este registro permite actuar sobre los timers para arrancarlos o pararlos. También posee indicadores de desbordamiento e indicadores de fuente de interrupción.

2.9.2 El timer 2

Este timer no se encuentra en el 8051. Se trata de un temporizador/contador de 16 bits con un prescaler programable con factores de división de 1, 2, 4 u 8. El timer puede ser programado en tres modos diferentes: desconectado, dependiente de la señal de reloj del sistema (temporizador) o como contador con una entrada de reloj externa. Es posible habilitar dos tipos de interrupción independientes: una para el desbordamiento de 8 bits y otra para el desbordamiento de 16 bits. Señalar que el timer 2 no es cargable con valores desde software (ver tabla 2.12).

Bit	Símbolo	Función
7	T2IS1	Para seleccionar la interrupción de desbordamiento de 16 bits
6	T2IS0	Para seleccionar la interrupción de desbordamiento de byte
5	T2ER	Habilitar reset externo (pin RT2)
4	T2B0	Bandera de interrupción de desbordamiento de byte
3	T2P1	Prescaler. 00:reloj,
2	T2P0	01:1/2 reloj 10:1/4 reloj, 11:1/8 reloj
1	T2MS1	Modo. 00:detenido, 01:reloj=1/12 f_{clk},
0	T2MS0	10:configuración reservada 11:reloj=pin T2

Tabla 2.12: El SFR TM2CON (dirección 0xEA)

Existen 4 registros de captura y 3 registro de comparación para el timer 2. Los registros de captura permiten, mediante una señal externa hacer que el 80592 cargue en ellos la cuenta del timer. Los registros de captura son CT0, CT1, CT2 y CT3, y están asociados a los pines de entrada CT0I a CT3I, respectivamente (ver tabla 2.13). Asímismo es posible también generar una interrupción en el momento en el que se realiza una captura (ver tabla 2.14).

Bit	Símbolo	Función	
		Captura	Evento de interrupción
7	CTN3	CT3I	Flanco de bajada
6	CTP3		Flanco de subida
5	CTN2	CT2I	Flanco de bajada
4	CTP2		Flanco de subida
3	CTN1	CT1I	Flanco de bajada
2	CTP1		Flanco de subida
1	CTN0	CT0I	Flanco de bajada
0	CTP0		Flanco de subida

Tabla 2.13: El SFR CTCON (0xEB)

El 80592 compara, en todo momento, el valor del timer 2 con los valores de los registros de comparación CM0, CM1 y CM2. En el caso de que haya una coincidencia entre el valor

Bit	Símbolo	Flags de interrupción
7	T2OV	Desbordamiento de 16 bits
6	CMI2	Captura en CM2
5	CMI1	Captura en CM1
4	CMI0	Captura en CM0
3	CTI3	Comparación en CT3
2	CTI2	Comparación en CT2
1	CTI1	Comparación en CT1
0	CTI0	Comparación en CT0

Tabla 2.14: El SFR TM2IR (0xC8)

del timer y el valor de alguno de estos registros, se pueden activar interrupciones y/o activaciones de pines en el puerto P4: el registro de comparación CM0 permite poner a 1 los pines P4.0 a P4.5, el registro de comparación CM1 permite poner a 0 los pines P4.0 a P4.5, y el registro de comparación CM2 permite conmutar de estado los pines P4.6 y P4.7.

Los otros dos SFR que afectan al timer 2 se encargan de controlar el funcionamiento de los 3 registros de comparación anteriormente citados. Éstos son el STE (dirección 0xEE, ver tabla 2.15) y el RTE (dirección 0xEF, ver tabla 2.16).

Bit	Símbolo	Función
7	TG47 (sólo lectura)	1: P4.7:=0 en la próxima coincidencia de CM2 0: P4.7:=1 en la próxima coincidencia de CM2
6	TG46 (sólo lectura)	1: P4.6:=0 en la próxima coincidencia de CM2 0: P4.6:=1 en la próxima coincidencia de CM2
5	SP45	1: P4.5:=1 cuando haya coincidencia de CM0
4	SP44	1: P4.4:=1 cuando haya coincidencia de CM0
3	SP43	1: P4.3:=1 cuando haya coincidencia de CM0
2	SP42	1: P4.2:=1 cuando haya coincidencia de CM0
1	SP41	1: P4.1:=1 cuando haya coincidencia de CM0
0	SP40	1: P4.0:=1 cuando haya coincidencia de CM0

Tabla 2.15: El SFR STE (0xEE)

2.9.3 El timer watch dog

En el 80592 existe un cuarto timer especial, denominado de watch dog. Este timer, cuando se desborda provoca un reset del microcontrolador y, por tanto, el software debe cargarlo regularmente. Su función es la de evitar que el microcontrolador entre en algún bucle infinito o en alguna sección de código que lo bloquee.

Bit	Símbolo	Función
7	TP47	1: P4.7 bascula cuando haya coincidencia de CM2
6	TG46	1: P4.6 bascula cuando haya coincidencia de CM2
5	RP45	1: P4.5:=0 cuando haya coincidencia de CM1
4	RP44	1: P4.4:=0 cuando haya coincidencia de CM1
3	RP43	1: P4.3:=0 cuando haya coincidencia de CM1
2	RP42	1: P4.2:=0 cuando haya coincidencia de CM1
1	RP41	1: P4.1:=0 cuando haya coincidencia de CM1
0	RP40	1: P4.0:=0 cuando haya coincidencia de CM1

Tabla 2.16: El SFR RTE (0xEF)

Mediante el pin /EW podemos configurar el modo de funcionamiento del micro desde el exterior. Si ponemos este pin a 0 habilitaremos el WDT (watch dog), mientras que si lo ponemos a 1, deshabilitaremos el watch dog y entraremos en modo power-down.

En el modo power-down, el procesador se para, manteniendo el contenido de la RAM interna intacto. El procesador sólo sale del modo power-down si se resetea o si recibe una interrupción de tipo wake-up del controlador CAN como aviso de que acaba de llegar un paquete por ese puerto. El modo power-down puede ser puesto tanto desde software (bit 1 del SFR PCON, en la dirección 0x87) o por hardware, poniendo a 1 el pin /EW del chip.

En circunstancias de operación normales, tendremos habilitado el WDT y deshabilitado el modo power-down (pin /EW a 0).

Para cargar el watch dog debemos, primero poner a 1 el bit 4 del SFR PCON y luego cargar el SFR del watch dog (dirección 0xFF) con el valor deseado.

2.10 Interrupciones

El 80592 permite manejar 15 vectores de interrupción, de los cuales los 5 primeros son heredados del 8051 (ver tabla 2.17). A cada interrupción se le asigna una prioridad. Los niveles de prioridad son dos: baja y alta.

Mediante los registros IEN0 e IEN1 habilitamos o deshabilitamos las interrupciones, mientras que con los registros IP0 e IP1 configuramos la prioridad de cada una de las interrupciones.

Las tablas 2.18 y 2.20 se corresponden con el registro de habilitación de interrupciones 0 y el registro de prioridad de interrupciones 0, que son registros presentes tanto en el 8051 como en el 80592. Los registros IEN1 (en la dirección 0xE8) e IP1 (en la dirección 0xF8) son exclusivos del 80592 y sólo afectan a los vectores de interrupción de este

Vector	Dirección	Procesador
Externa 0	0x0003	8051 y 80592
Desbordamiento del timer 0	0x000B	8051 y 80592
Externa 1	0x0013	8051 y 80592
Desbordamiento del timer 1	0x001B	8051 y 80592
Serial 0 (UART)	0x0023	8051 y 80592
Serial 1 (CAN)	0x002B	80592
T2 captura 0	0x0033	80592
T2 captura 1	0x003B	80592
T2 captura 2	0x0043	80592
T2 captura 3	0x004B	80592
Conversión ADC completada	0x0053	80592
T2 comparación 0	0x005B	80592
T2 comparación 1	0x0063	80592
T2 comparación 2	0x006B	80592
Desbordamiento del timer 2	0x0073	80592

Tabla 2.17: Vectores de interrupción del 8051 y del 80592

microcontrolador (ver las tablas 2.19 y 2.21).

Bit	Símbolo	Función
7	EA	Habilitación/inhabilitación global de todas las interrupciones
6	EAD	Habilitar interrupciones del ADC
5	ES1	Habilitar interrupciones del controlador CAN
4	ES0	Habilitar interrupciones de la UART
3	ET1	Habilitar interrupciones del timer 1
2	EX1	Habilitar interrupciones externas del pin /INT1
1	ET0	Habilitar interrupciones del timer 0
0	EX0	Habilitar interrupciones externas del pin /INT0

Tabla 2.18: El SFR IEN0 (dirección 0xA8)

2.11 La UART

El 80592 posee una interfaz de comunicaciones serie heredada del 8051. Se accede al puerto serie desde software mediante los SFR SBUF (dirección 0x99) y SCON (dirección 0x98). Los pines asociados al puerto serie son el RXD (bit 0 del puerto 3) y el TXD (bit 1 del puerto 3).

El SFR SCON permite configurar el puerto serie, mientras que el registro SBUF es el buffer de transmisión recepción de la UART. Una escritura en este SFR provoca una transmisión del dato, y una lectura lee el último dato recibido por la UART (ver tabla

Bit	Símbolo	Función
7	ET2	Habilitar interrupciones de desbordamiento del timer 2
6	ECM2	Habilitar interrupciones del comparador CM2 del timer 2
5	ECM1	Habilitar interrupciones del comparador CM1 del timer 2
4	ECM0	Habilitar interrupciones del comparador CM1 del timer 2
3	ECT3	Habilitar interrupciones del capturador CT3 del timer 2
2	ECT2	Habilitar interrupciones del capturador CT2 del timer 2
1	ECT1	Habilitar interrupciones del capturador CT1 del timer 2
0	ECT0	Habilitar interrupciones del capturador CT0 del timer 2

Tabla 2.19: El SFR IEN1 (dirección 0xE8)

Bit	Símbolo	Función
7	–	Sin usar
6	PAD	Nivel de prioridad del ADC
5	PS1	Nivel de prioridad del CAN
4	PS0	Nivel de prioridad de la UART
3	PT1	Nivel de prioridad del timer 1
2	PX1	Nivel de prioridad de /INT1
1	PT0	Nivel de prioridad del timer 0
0	PX0	Nivel de prioridad de /INT0

Tabla 2.20: El SFR IP0 (dirección 0xB8)

2.22).

El puerto serie puede funcionar en 4 modos diferentes:

- `Modo 0 (SM0 = 0, SM1 = 0)`. El puerto actúa como registro de desplazamiento, utilizando el pin RXD con entrada/salida de datos y el pin TXD como salida de reloj. f_{osc} / 12 bits por segundo.
- `Modo 1 (SM0 = 0, SM1 = 1)`. UART en modo full duplex (1 bit de start y otro de stop). 8 bits de datos. Bits por segundo variables (timer 1).
- `Modo 2 (SM0 = 1, SM1 = 0)`. UART en modo full duplex (1 bit de start y otro de stop). 9 bits de datos. f_{osc} / 64 ó f_{osc} / 32 bits por segundo.
- `Modo 3 (SM0 = 1, SM1 = 1)`. UART en modo full duplex (1 bit de start y otro de stop). 9 bits de datos. Bits por segundo variables (timer 1).

En modo 2 ó 3, si SM2 = 1 , RI sólo se pondrá a 1 si el 9º bit es un 1. En modo 1, si SM2 = 1, RI sólo se activará al recibirse un bit de stop. En modo 0, el bit SM2 debe estar a 0. El bit REN sirve para habilitar la recepción (con REN = 1). TB8 es el 9º bit a transmitir en los modos 2 ó 3, mientras que RB8 es el 9º bit recibido en los modos 2 ó 3. En modo 1, si SM2 = 0, RB8 es el bit de stop del paquete recibido.

Bit	Símbolo	Función
7	PT2	Nivel de prioridad del desbordamiento del timer 2
6	PCM2	Nivel de prioridad del comparador CM2
5	PCM1	Nivel de prioridad del comparador CM1
4	PCM0	Nivel de prioridad del comparador CM0
3	PCT3	Nivel de prioridad del capturador CT3
2	PCT2	Nivel de prioridad del capturador CT2
1	PCT1	Nivel de prioridad del capturador CT1
0	PCT0	Nivel de prioridad del capturador CT0

Tabla 2.21: El SFR IP1 (dirección 0xF8)

0x9F	0x9E	0x9D	0x9C	0x9B	0x9A	0x99	0x98
SM0	SM1	SM2	REN	TB8	RB8	TI	RI
Modo							

Tabla 2.22: El SFR SCON (0x98)

TI indica cuando se ha terminado una transmisión. Antes de volver a transmitir, debe ser puesto a 0 por software. El bit RI es el indicador de recepción, e indica cuando se ha recibido el último bit de un paquete. Debe también ser puesto a 0 por el software.

2.12 El controlador CAN incluido

La conexión con el bus CAN el la característica más compleja de este microcontrolador. Las entradas CAN del 80592 se localizan en los pines CRX0 y CRX1, mientras que las salidas CAN forman parte del puerto P1 (CTX0, se corresponde con el bit 6, y CTX1 con el bit 7).

2.12.1 El protocolo CAN

El protocolo CAN (Control Area Network) es un protocolo serie que, al igual que el Ethernet, se basa en el uso de un par de cables con terminaciones de impedancia fija en los extremos y al que se pueden conectar nodos en cualquier punto de su longitud. La impedancia característica en los buses CAN suele ser de 124 Ω. El protocolo CAN es un protocolo de tipo CSMA/CD (Carrier Sense Multiple Access / Collision Detection), es decir; un protocolo de acceso múltiple, en el que cada nodo es capaz de detectar una transmisión cualquiera y además es capaz de detectar colisiones entre paquetes de bits que se estén enviando simultáneamente sobre el bus. Esto es, sólo es posible un paquete ocupando el cable en cada instante de tiempo.

En la nomenclatura CAN no se utilizan los términos "0"y "1"para indicar el estado del bus en un instante determinado. Se utilizan las nociones de bit "dominante"y bit "recesivo".

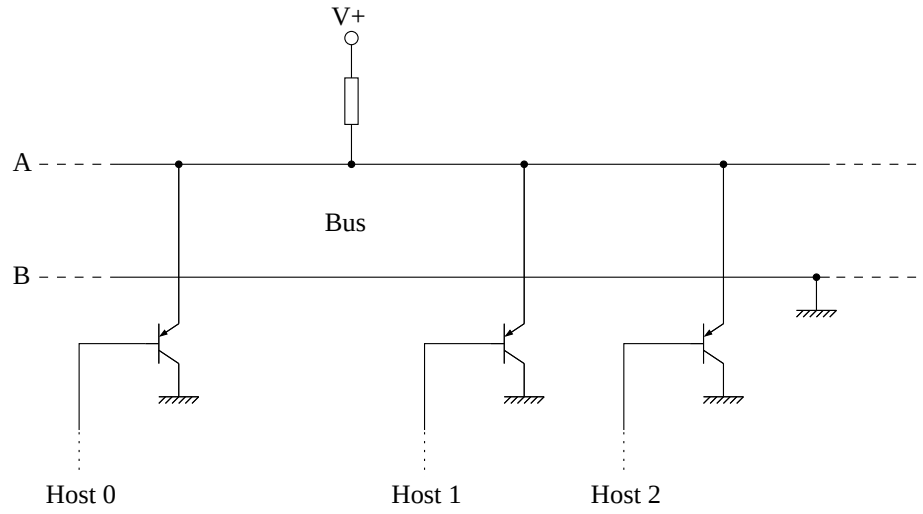


Figura 2.5: Modelo de transmisión sobre un bus basado en una AND cableada

El bus CAN utiliza un sistema para transportar datos muy sencillo: una AND cableada (ver figura 2.5). Esto es, sólo si TODOS los nodos conectados al bus están en estado recesivo (a "1"), el bus se encontrará es estado recesivo; sin embargo, con que un sólo nodo envíe un bit dominante al bus (un "0"), éste pasará a estado dominante, independientemente de que otros nodos estén enviando otros bits recesivos. De este fenómeno proviene el nombre de "dominante"y "recesivo"para los bits 0 y 1 del bus, respectivamente.

En el ejemplo de la figura 2.5, si todos los hosts envían un "1"(valor alto), todos los transistores permanecen en corte y, por tanto el bus mantiene un voltaje V entre los puntos A y B. En cambio, si uno de los hosts envía un "0", este cero provocará que el transistor correspondiente se ponga en saturación, cortocircuitando las líneas A y B; sin importar lo que el resto de hosts esté enviando.

Actualmente existen en la industria dos modalidades de protocolo CAN: CAN2.0A y CAN2.0B. Las diferencias entre una y otra son mínimas, permitiendo la versión B, direccionar mayor cantidad de nodos en una red. El 80592 implementa el CAN2.0A.

Dentro del protocolo CAN tenemos 4 tipos de paquetes de datos:

- Paquetes de datos, para transportar datos.
- Paquetes de solicitud, para solicitar datos.
- Paquetes de error, para indicar errores en la red.
- Paquetes de sobrecarga (no soportador por el 80592).

Un paquete de datos se compone de varios campos:

Inicio de frame	Un bit dominante para indicar inicio de frame.
Campo de arbitraje	11 bits con el identificador del paquete más un un bit RTR (dominante) (en la variante CAN2.0B del protocolo el identificador es de 29 bits).
Campo de control	6 bits, los dos primeros dominantes, y los 4 restantes indicando el número de bytes en el campo de datos.
Campo de datos	Entre 0 y 8 bytes de longitud. Para cada byte se envía primero el bit más significativo.
Campo de CRC	15 bits con el CRC más un bit recesivo.
Campo ACK	2 bits recesivos.
Fin de frame	7 bits recesivos.

Tanto los 11 bits del indentificador del paquete como los 4 bits que indican el tamaño del campo de datos se transmiten empezando por el bit más significativo. El campo identificador NO puede tener los siete bits más significativos (ID.10 a ID.4) todos dominantes.

De los dos bits recesivos del campo ACK, el primero está destinado para que el nodo que está recibiendo el paquete, lo rellene con un bit dominante, tras comprobar los datos y el campo CRC. El nodo transmisor que, mientras transmite, monitoriza la red, sabe si el paquete ha llegado bien a su destino comprobando que la línea, en el primer bit del campo ACK, se pone en estado dominante.

Algo parecido ocurre con el campo RTR. Cuando un nodo desea solicitar datos a otro envía un paquete con el bit RTR recesivo (un paquete de solicitud). El nodo que responde a la solicitud lo hace enviando un paquete de datos con el mismo campo identificador que el paquete de solicitud, pero con este bit en estado dominante. Si en el mismo instante que el nodo solicitante envía su paquete, el nodo con los datos procede a enviar los datos, no ocurre colisión alguna, al utilizar ambos nodos el mismo identificador y al transformarse, debido a la emisión del bit dominante por parte del nodo con los datos, el bit RTR a 0.

El campo identificador en un paquete CAN, identifica a ese paquete, no al nodo que lo envía o recibe. Un nodo puede enviar diferentes paquetes con diferentes campos ID en cada uno de ellos y también recibir los paquetes que crea conveniente. Lo normal es que se utilize el sistema indicado en el ejemplo anterior: un nodo A envía un paquete de solicitud de datos con el identificador=x y otro nodo B, que se encuentra a la escucha de paquetes con ese identificador, responde con un paquete de datos y el campo identificador= x.

Un paquete de solicitud de datos se compone de los siguientes campos:

Inicio de frame	Un bit dominante para indicar inicio de frame.
Campo de arbitraje	11 bits con el identificador del paquete más un un bit RTR (recesivo).
Campo de control	6 bits, los dos primeros dominantes, y los 4 restantes se ignoran.
Campo de CRC	15 bits con el CRC más un bit recesivo.
Campo ACK	2 bits recesivos.
Fin de frame	7 bits recesivos.

Un paquete de error contiene dos campos: La superposición de indicadores de error (bits dominantes), seguido del delimitador de error.

Un nodo CAN puede configurarse como activo o pasivo ante los errores. En el caso de que un nodo sea "Error Active"y detecte alguna condición de error en el bus, transmitirá 6 bits dominantes consecutivos como indicador de error [1]. El resto de nodos de la red detectarán esta condición de error y enviarán sus respectivos indicadores de error sobre el bus. Los nodos configurados como "Error Pasive"envían 6 bits recesivos consecutivos.

Tras la superposición de indicadores de error, y la detección por parte de todos los nodos de la condición de error, esperarán a que haya una transición de dominante a recesivo en el bus y dejarán 8 bits recesivos como delimitador del error.

2.12.2 Conexionado del 80592 con un bus CAN

El micro utilizado posee un controlador de bus CAN incorporado. Este controlador no posee, sin embargo, características de *transceptor*; es decir, no podemos conectar directamente las patitas del bus CAN del 80592 a un bus, es preciso acondicionar la señal y adaptar impedancias.

Es por esto por lo que se hace necesario el uso del 82250. La conexión se realiza de forma directa (ver la figura 2.6) y sólo hay que tener en cuenta desde el punto de vista del hardware, el respetar la impedancia del bus y en adecuar las pendientes de transición a la hora de transmitir (de 0 a 1 y de 1 a 0). Para controlar estas pendientes se ha situado el potenciómetro en el pin R_s del transceptor 82250. Si colocamos el dial del potenciómetro en el extremo de 0 voltios, obtenemos unas transiciones en la transmisión lo mas rápidas posible (para buses de alta velocidad), mientras que jugando con valores intermedios obtendremos pendientes de transición más acordes con el tipo de cable que tengamos, el alcance, etc (buses de media y baja velocidad).

En la figura 2.8 se puede apreciar una fotografía del microcontrolador junto al transceptor CAN.

[1] Estos 6 bits dominantes consecutivos permiten detectar un error por parte de cualquier nodo de la red, ya que es imposible confundirlo con un paquete de datos o de solicitud, a tener estos paquetes la restricción de que los 7 bits más significativos del campo ID no pueden ser todos dominantes

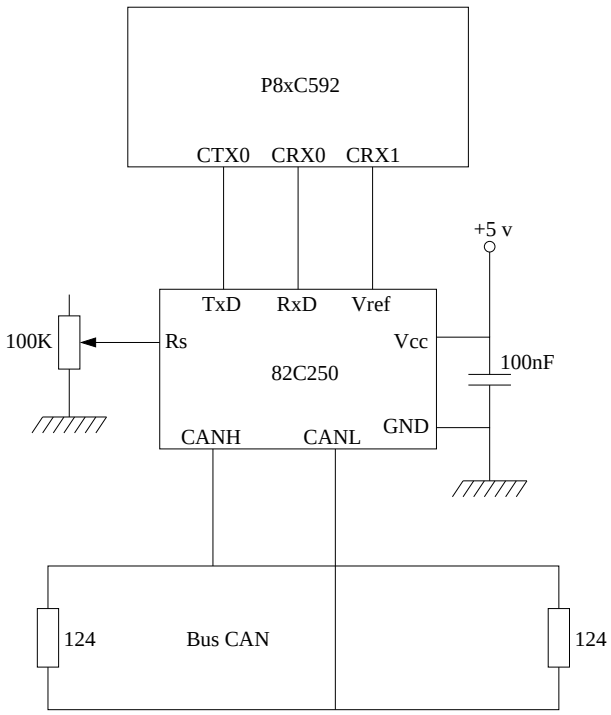


Figura 2.6: Conexionado entre el 80592 y el transceptor CAN 82250

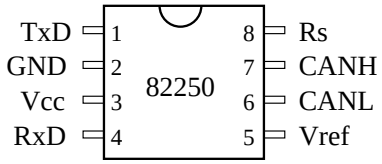


Figura 2.7: Encapsulado del transceptor CAN 82250

2.12.3 Programación del bus CAN en el 80592

El acceso al controlador CAN en el 80592

El microcontrolador 80592 permite el acceso al controlador CAN incluido en el chip a través de los SFR:

- `CANSTA (en 0xD8)`: Para leer el status del controlador y escribir direcciones para hacer transferencias DMA entre el controlador CAN y la RAM principal.
- `CANCON (en 0xD9)`: lectura de los flags de interrupción y escritura de comandos al controlador CAN.
- `CANDAT (en 0xDA)`: Para leer y escribir datos en un registro del controlador CAN.
- `CANADR (en 0xDB)`: Para seleccionar el registro del controlador CAN al que queremos acceder a través del registro CANDAT.

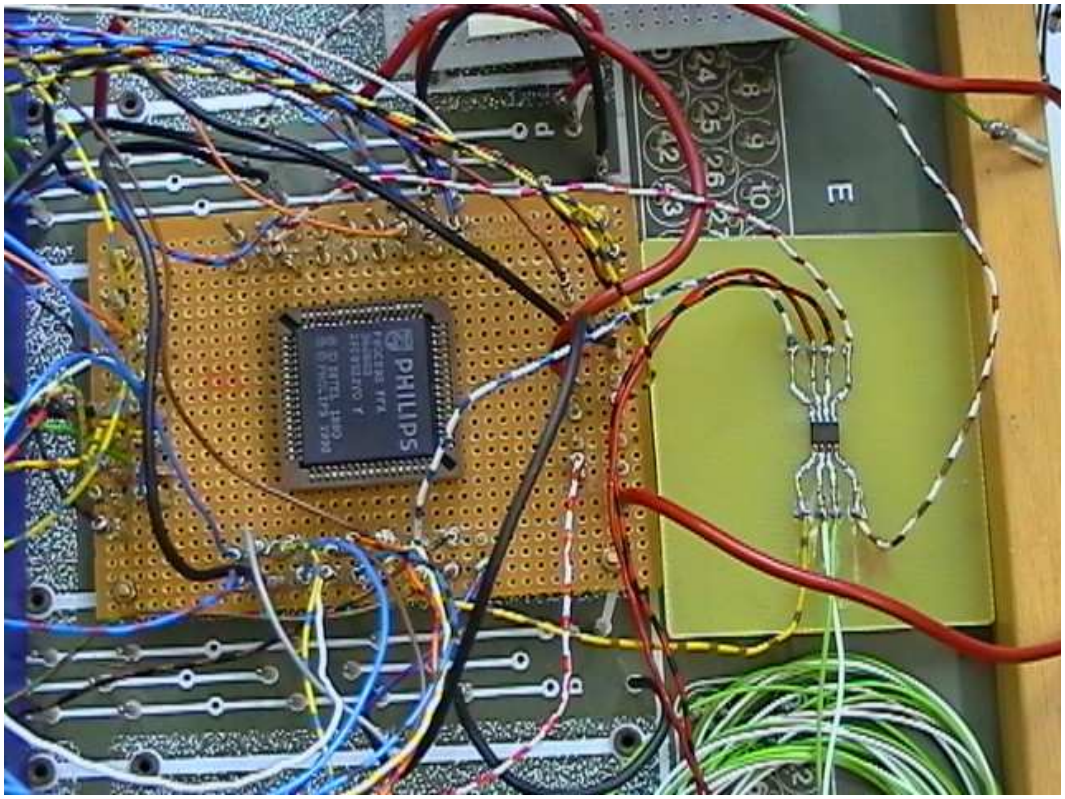


Figura 2.8: Detalle del 80592 (izquierda) conectado al 82250 (derecha)

Por otro lado tenemos los registros del propio controlador CAN:

- `0x00: Control Register (CR)`. Habilitación de interrupciones en el lado del controlador, voltajes de referencia, etc.
- `0x01: Command Register (CMR)`. Configuración de los pines de recepción CAN (CRX0 y CRX1) del 80592, comandos especiales para borrado de buffers, modos de operación, etc.
- `0x02: Status Register (SR)`. Contiene los flags que indican el estado del controlador CAN.
- `0x03: Interrupt Register (IR)`. Registro de sólo lectura que permite a la rutina de servicio de interrupción que atiende las interrupciones del bus CAN, identificar exactamente qué tipo de interrupción se ha generado (recepción, transmisión, error, etc).
- `0x04: Acceptance Code Register (ACR)`. Permite indicar qué tipo de mensajes CAN se van a aceptar en el buffer de recepción (se compara este registro con los 8 bits más significativos del campo ID del paquete CAN).
- `0x05: Acceptance Mask Register (AMR)`. Permite definir qué bits del registro ACR son *Don't care*.

- `0x06: Bus Timming Register 0 (BTR0)`. Registro para el control del timming del bus CAN.
- `0x07: But Timming Register 1 (BTR1)`. Segundo registro para el control del timming del bus CAN.
- `0x08: Output Control Register (OCR)`. Permite configurar las características eléctricas de las salidas CTX0 y CTX1 del bus CAN.
- `0x0A a 0x13`. El buffer de transmisión. Aquí el software escribe el paquete CAN que desea transmitir.
- `0x14 a 0x1D`. El buffer de recepción. Aquí el software lee los paquetes CAN entrantes. El 80592 posee 2 buffers de recepción configurados a modo de cola, este sistema de doble buffer queda oculto al usuario; de tal forma que si tenemos 2 paquetes diferentes en los buffers de recepción en una primera lectura desde el software leeremos el primer paquete en llegar mientras que con una segunda lectura leeremos el segundo (no hay que indicarle al controlador cual es el buffer que queremos leer, ya que se utiliza un sistema de colas: se lee primero el primero en llegar).

Si queremos, por ejemplo, acceder al registro CMR (`Command Register`), escribiremos primero en el SFR CANADR el valor 0x01 (el índice del registro CMR en el controlador CAN) y a través del SFR CANDAT ya podremos leer y/o escribir en el registro seleccionado.

Programación del timming en el 80592

La sección del timming del bus CAN y de su programación constituyen un tema un tanto oscuro en la hoja de datos del 80592. Gracias a algunas notas de aplicación encontradas en internet ha sido posible obtener las fórmulas para el cálculo del timming y para programar los registros BTR0 y BTR1.

$$f_{bit} = \frac{f_{xtal}}{(3 + TSEG1 + TSEG2 \pm (SJW + 1)) \cdot (BRP + 1) \cdot 2}$$

Esta fórmula relaciona la frecuencia de bit del bus CAN con la frecuencia del cristal de cuarzo del microcontrolador. Los valores TSEG1, TSEG2, SJW, etc son campos dentro de los registros BTR0 y BTR1 del controlador CAN.

Como se puede apreciar, en el demonimador tenemos una operación $\pm$. En el caso del timming en el bus CAN podemos definir una "olgura" la hora de calcular el número de bits por segundo. Esta olgura permitirá a los circuitos de recepción ser más permisivos y poder recibir paquetes CAN que no se ajusten exactamente al timming definido.

Pongamos un ejemplo:

$$\begin{aligned} f_{xtal} &= 16Mhz \\ TSEG1 &= 3 \\ TSEG2 &= 4 \\ SJW &= 1 \\ BRP &= 23 \\ f_{bit} &= \frac{16000000}{(3+3+4\pm(1+1))\cdot(23+1)\cdot 2} \\ f_{bit} &= \frac{16000000}{(10\pm 2)\cdot 24\cdot 2} \\ f_{bit} &= \frac{16000000}{(10\pm 2)\cdot 48} \end{aligned}$$

Si calculamos las dos ecuaciones resultantes de la ecuación anterior, en una utilizando el $+$ del $\pm$ y en la otra utilizando el $-$ del $\pm$, obtenemos:

$$\begin{aligned} f_{bitmax} &= 41666'667bps \\ f_{bitmin} &= 27777'778bps \end{aligned}$$

Estas dos frecuencias de bit determinan la frecuencia máxima y mínima a la que el controlador CAN puede recibir datos del bus. La frecuencia a la que el controlador transmite es:

$$f_{bit} = \frac{f_{xtal}}{(3+TSEG1+TSEG2)\cdot(BRP+1)\cdot 2}$$

Nótese se se ha eliminado la operación $\pm$ sobre el valor $(SJW+1)$. En este caso, la velocidad de transmisión será:

$$f_{bit} = 33333'333bps$$

El registro BTR0:

SJW_1	SJW_0	BRP_5	BRP_4	BRP_3	BRP_2	BRP_1	BRP_0

El registro BTR1:

SAM	$TSEG2_2$	$TSEG2_1$	$TSEG2_0$	$TSEG1_3$	$TSEG1_2$	$TSEG1_1$	$TSEG1_0$

El bit SAM del registro BTR1 permite indicar la cantidad de veces que se muestrea una ranura de tiempo de un bit para determinar si está a 0 o a 1. Si SAM=0 en cada ranura de tiempo se muestrea una sola vez para determinar el bit, mientras que si SAM=1 en cada ranura de tiempo se muestrea 3 veces, de esta forma se determina si se trata de un 0 o un 1 por mayoría.

El uso de un sólo muestreo (SAM=0) se utiliza en buses de alta velocidad, el uso de 3 muestreos (SAM=1) se utiliza en buses de baja y media velocidad (menos de 1MHz). En nuestro caso, el montaje del bus se ha realizado en base a un par trenzado con lo que es acosejable utilizar una baja velocidad de transferencia.

Capítulo 3

El sistema de desarrollo básico

El sistema básico consta de los componentes básicos para el funcionamiento del microcontrolador.

- *Flash ROM* : Una memoria ROM modelo 29F010 de 128 Kbytes.
- *RAM* : Una memoria RAM estática modelo 62256 de 32 Kbytes.
- *Microcontrolador* : MCU 80592.
- *DAC* : Un conversor digital-analógico externo.

Mediante estos circuitos periféricos al 80592 nos será posible desarrollar software de forma cómoda, mediante el uso de la memoria FlashROM, y dotar a nuestro procesador de una salida analógica gracias al DAC externo. Con la incorporación de una pastilla RAM externa aumentamos la memoria de datos del microcontrolador hasta 32 Kbytes.

3.1 Alimentación

El microcontrolador posee múltiples entradas de alimentación para cada una de sus partes (ver tabla 3.1).

Se debe cumplir que $AV_{ref+} > AV_{ref-}$ y, además, $AV_{ref+} < AV_{dd}$ y $AV_{ref-} > AV_{ss}$.

En nuestro caso: $V_{dd} = AV_{dd} = AV_{ref+} = 5$ voltios y $V_{ss} = AV_{ss} = CV_{ss} = AV_{ref-}$ = Masa. El pin REF puede ser configurado como entrada, como salida o sin uso: si se configura como salida, lo que saca es un voltaje de 1 / 2 AV_{dd} con el objetivo de que sea utilizado por un transceptor del bus CAN como voltaje de refencia para enviar los 0s y 1s recibidos al 80592 a través de las entradas CRX0 y/o CRX1. Si se configura como entrada, es el transceptor del bus CAN el que debe generar un voltaje de referencia para

V_{dd}	Alimentación de la parte digital (+5 voltios).
V_{ss}	Masa (0 voltios) de la parte digital.
AV_{dd}	Alimentación del ADC y la parte CAN (+5 voltios).
AV_{ss}	Masa (0 voltios) del ADC y la parte CAN.
CV_{ss}	Potencial de tierra para las salidas CAN.
AV_{ref-}	Valor de voltaje para la configuración 0x000 en el ADC.
AV_{ref+}	Valor de voltaje para la configuración 0x3FF en el ADC.
REF	Entrada/salida opcional con el valor de voltaje de referencia para leer los valores de las entradas CAN CRX0 y CRX1.

Tabla 3.1: Pines de alimentación del microcontrolador

que los comparadores del controlador CAN del 80592 puedan leer adecuadamente los bits que le llegan a través de CRX0 y CRX1. La otra forma de configurarlo es ni como entrada ni como salida, en cuyo caso se interpretan los bits como sigue:

- Si nivel CRX0 > nivel CRX1 $\Longrightarrow$ bit recesivo.
- Si nivel CRX0 < nivel CRX1 $\Longrightarrow$ bit dominante.

Si se configura el pin REF como pin sin utilizar (tal y como se ha puesto en el prototipo) debe ir conectado, mediante un condensador > 1 nF a AV_{ss}.

3.2 Circuito de reloj

Para generar la señal de reloj para el 80592 existen dos alternativas, según la documentación de Philips:

- Utilizar un cristal de cuarzo.
- Utilizar un reloj externo.

En este caso se ha optado por la utilización de una fuente de reloj externa, ya que al utilizar un cristal hubo problemas de distorsión en la onda cuadrada. Se utilizó un reloj a cuarzo ya encapsulado y que genera una señal muy estable y con poca distorsión.

La señal de reloj se introduce directamente al pin XTAL1, dejando el pin XTAL2 sin conectar. El oscilador genera una onda de 1.8 MHz, aunque es posible utilizar un cristal de hasta 16 Mhz.

3.3 Circuito de reset

El circuito de reset montado alrededor del pin RST del microcontrolador es el habitual en este tipo de montajes. Lo que se ha puesto es un condensador entre el pin de reset del 80592 y V_{dd}, de tal forma que al suministrar tensión a todo el prototipo el microcontrolador se reinicie (ver figura 3.1).

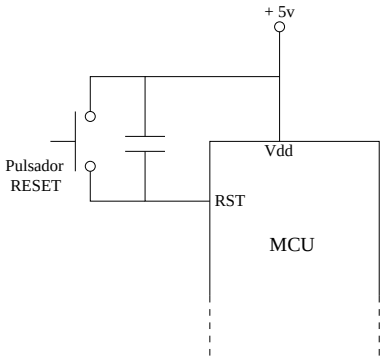


Figura 3.1: Circuito de reset

Además de este reset de arranque, se ha añadido un pulsador en paralelo al condensador para que el usuario pueda, en cualquier momento reiniciar el procesador sin que por ello se pierda, por ejemplo, el contenido de la RAM externa o sin necesidad de cortar la energía.

3.4 La memoria Flash

Se trata de una memoria de 1 Mbit organizada en 131072 direcciones de 8 bits cada una (128 Kbytes) en la que estará ubicado el código del programa. Aparte de las patillas de datos (D0 a D7) y de direcciones (A0 a A16), el chip posee los habituales pines de control.

`/WE`	Entrada que se utiliza para programar y borrar la flash, en el montaje del prototipo se pone a 1 (V_{cc}) para evitar que su contenido se borre o reprograme por accidente.
`/OE`	Un flanco de bajada emite por el bus de datos (D0 a D7) el contenido de la dirección apuntada por las entradas (A0 a A16); si se pone a 1, las salidas de datos pasan a alta impedancia.
`/CE`	*Chip Enable*. Cuando se pone a 0 habilita el chip, cuando está a 1, todas las entradas y salidas del integrado permanecen en alta impedancia.

Los pines V_{cc} y V_{ss} son las entradas de alimentación a la pastilla (V_{cc} = 5 voltios, V_{ss} = 0 voltios).

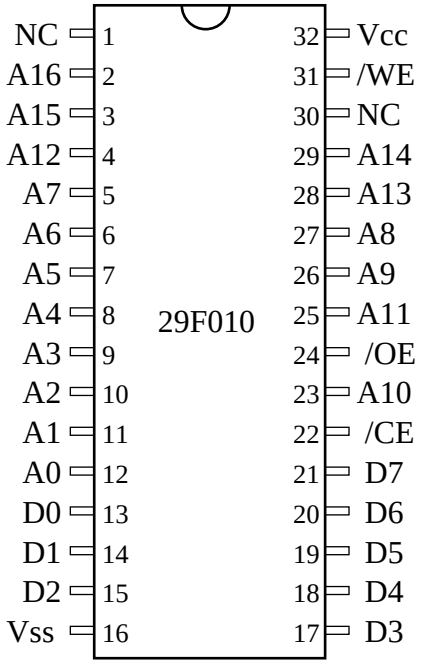


Figura 3.2: Encapsulado del 29F010 (FlashROM)

3.5 La RAM

La memoria RAM es de tipo estático de 256 Kbits organizada en 32768 direcciones de 8 bits cada una (32 Kbytes), este RAM se utilizará como memoria de datos externa. En la figura 3.3 se muestra el encapsulado de este chip.

Los pines son los habituales en cualquier pastilla de RAM estática:

`/WE` Un flanco de bajada en este pin provoca que se escriba en la dirección de memoria seleccionada (A0 a A14) los 8 bits de datos presentes en los pines de datos (D0 a D7). Los pines de datos actuan como entradas.

`/OE` Un flanco de bajada emite por el bus de datos (D0 a D7) el contenido de la dirección apuntada por las entradas (A0 a A14). Los pines de datos actuan como salidas.

`/CS` *Chip Select*. Cuando se pone a 0 habilita el chip tanto para leer como para escribir. Puesto a 1, pone todas los pines del integrado en alta impedancia.

Nótese que la configuración /WE=0 y /OE=0 es una configuración ilegal.

Al igual que en el caso de la memoria flash, los pines V_{cc} y V_{ss} son las entradas de alimentación de la pastilla (V_{cc} = 5 voltios, V_{ss} = 0 voltios).

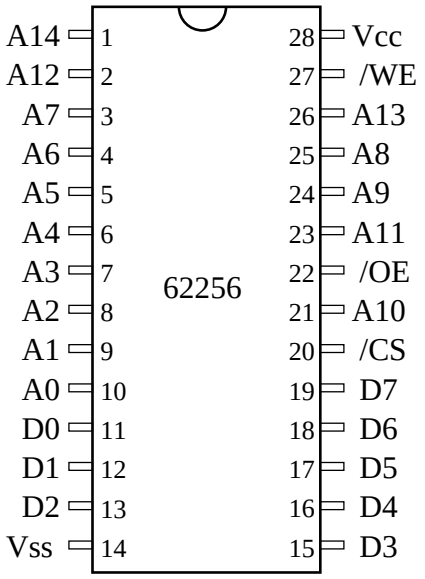


Figura 3.3: Encapsulado del 62256 (RAM estática)

3.6 Conexionado de las memorias al 80592

En al figura 3.4 podemos ver la forma en que hemos demultiplexado el bus de datos y de direcciones mediante una pastilla 74573, utilizando la salida ALE del microcontrolador.

La parte baja del bus de direcciones y el bus de datos salen/entran del/al puerto P0 en momentos de tiempo diferentes (primero la parte baja de la dirección y luego los datos). El latch triestado 74573 permite demultiplexar el puerto P0 y obtener así los dos buses.

Cuando el 80592 va a realizar un acceso a las memorias externas, éste se realiza en varios pasos:

1. Emite la parte alta de la dirección por el puerto P2.
2. Emite la parte baja de la dirección por el puerto P0.
3. Emite un flanco de subida por el pin ALE.
4. Vuelve a poner el pin ALE a 0.
5. Emite los datos por el puerto P0 si va a escribir o lo pone en modo entrada si va a leer.
6. Activa el bus de control de las memorias externas:
   - Si va a leer una instrucción emite un flanco de bajada por /PSEN.
   - Si va a leer un dato emite un flanco de bajada por /RD.
   - Si va a escribir un dato emite un flanco de bajada por /WR.

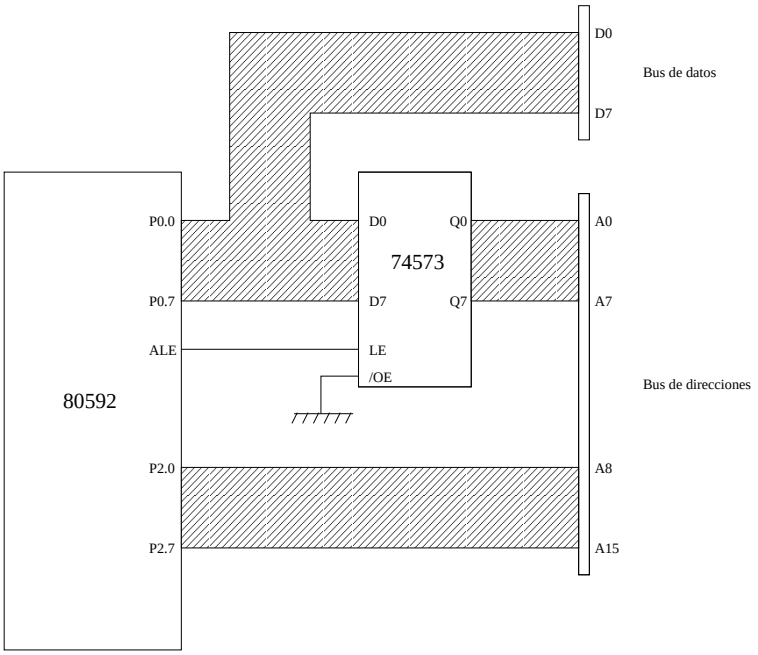


Figura 3.4: Demultiplexación de los buses del 80592

7. Si se trata de una lectura, procede a leer el/la dato/instrucción del puerto P0.

El pin ALE, por tanto lo podemos conectar directamente a la entrada LE del 74573 (al activarse ésta mediante flanco de subida). Así tenemos que la parte baja del bus de direcciones sale por el 74573 y el bus de datos lo podemos coger sin ningún problema directamente de P0, ya que los datos salen después de la parte baja de la dirección y para entonces ya tenemos "recordada"la dirección en el latch.

De esta forma podemos conectar fácilmente una RAM de datos y una ROM de programa externas a nuestro microcontrolador. Mediante las líneas /RD y /WR se accede a la RAM externa mientras que con la línea /PSEN se solicita una instrucción (lectura de la memoria de programa (ROM)). Obtenemos, finalmente una arquitectura pseudo-Harvard para nuestro sistema (ver figura 3.6).

3.7 Conexión de un conversor DA al prototipo

El microcontrolador 80592, como la mayoría de los microcontroladores, no posee un DAC interno. Para dotar de mayor versatilidad a nuestro prototipo, le hemos incorporado un DAC externo en el puerto P4 del micro (ver figura 3.7). De esta manera disponemos de una salida analógica, muy útil para la implementación de un lazo de control.

De esta forma, al escribir un dato en el puerto P4 (dirección 0xC0 en los SFRs) obtenemos un voltaje entre $-V_{ref}$ (configuración binaria 0) y $+V_{ref}$ (configuración binaria 255) a la salida del conversor.

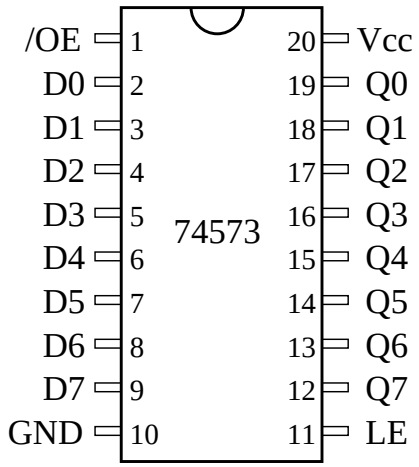


Figura 3.5: Encapsulado del 74573 (latch triestado)

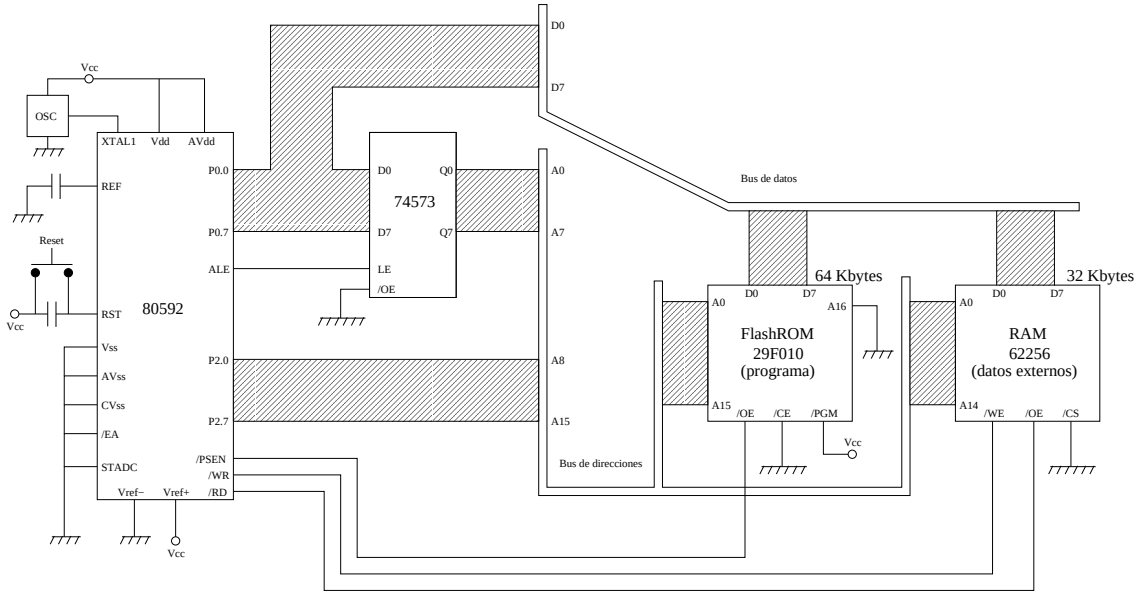


Figura 3.6: El sistema básico

Gracias a esta salida analógica que posee el prototipo es muy sencillo integrarlo en una planta real.

En la figura 3.10 podemos apreciar en una placa de prototipos: en la zona superior izquierda el pulsador de reset seguido del oscilador de cuarzo, más a la derecha tenemos el 74573 junto a la pastilla de FlashROM 29F010. El la zona inferior derecha tenemos la RAM externa (62256) y en la zona inferior izquierda tenemos el conversor digital-analógico.

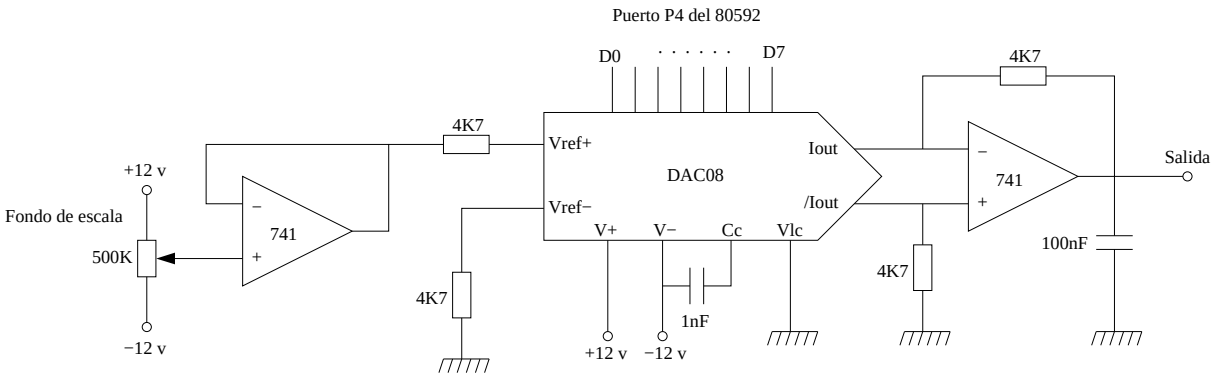


Figura 3.7: Esquema del montaje del DAC con el puerto P4

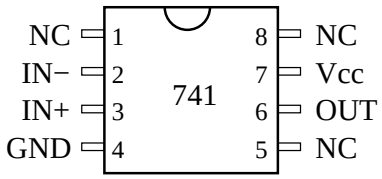


Figura 3.8: Encapsulado del 741 (amplificador operacional)

3.8 Programación del prototipo

La parte software corre a cargo del compilador SDCC (*Small Device C Compiler*) un compilador ANSI C desarrollado bajo licencia GNU GPL que permite generar código para la familia MCS51, entre otros procesadores[1]. Este compilador genera ficheros de tipo Intel Hex (extensión `.ihx`) que, mediante un programador de FlashROM, se pasan al chip 29F010 para su posterior ejecución en el prototipo.

En el compilador SDCC la sintaxis ANSI C ha sido ampliada con nuevas palabras reservadas y que permiten realizar funciones específicas para microcontroladores MCS-51:

```
sfr at 0x80 reg;    /* Para definir SFRs */
sbit at 0xD7 cy;    /* Un bit en el espacio de los SFRs */
data int x;         /* Un entero (16 bits) en la RAM interna */
idata int w;        /* Un entero (16 bits) en la RAM interna de acceso
                       indirecto */
xdata long int y;   /* Un entero largo (32 bits) en la RAM externa */
bit bandera;        /* Una variable de 1 bit en el espacio de usuario */
```

Es posible, además, utilizar direccionamiento absoluto (`at N`) para alojar las variables, aunque no sean de tipo `sfr` o `sbit`:

```
xdata at 0x8000 unsigned char i;
```

[1]El compilador SDCC puede ser descargado tanto en su versión estable (.tar.gz) como a través del CVS de `http://sdcc.sourceforge.net`.

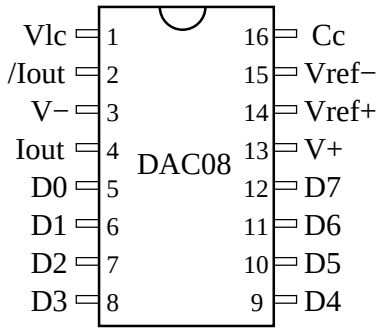


Figura 3.9: Encapsulado del DAC08 (DAC)

```
bit at 0x02 un_bit;
```

Hay también palabras reservadas para manipular interrupciones, para manipular los parámetros y la pila para aprovechar memoria, para la utilización de los bancos de registros (en 80592 posee 4 bancos en la parte baja de la RAM interna, cada banco, a su vez tiene 8 registros R0 a R7, y sólo puede haber un banco activo al mismo tiempo), etc. Para más información, acudir al apéndice A.

Existen, además, ciertas cuestiones a la hora de programar el 80592 desde C que deben ser tenidas en cuenta:

- El timer watch dog del 80592.
- Los puertos adicionales para el ADC, el bus CAN, el PWM, etc.
- Vectores de interrupción adicionales a los del 8051.

3.8.1 El timer watch dog del 80592

El 80592 posee dos modos de funcionamiento seleccionables mediante el pin /EW del micro (Enable Watchdog). Cuando este pin se pone a 1, el procesador entra en modo power-down, mientras que si está a 0 se habilita el timer de watch dog. Este timer es un contador independiente que, cuando se pone a 0, tras un desbordamiento, reinicia el procesador. En nuestro caso el procesador tiene este pin permanentemente a 0.

La frecuencia del timer de watch dog depende de la frecuencia de reloj del procesador y se calcula mediante la fórmula:

$$f_{timer} = \frac{f_{CLK}}{2 \cdot 2048}$$

Es por esto por lo que el software debe, regularmente, reiniciar el contador del timer

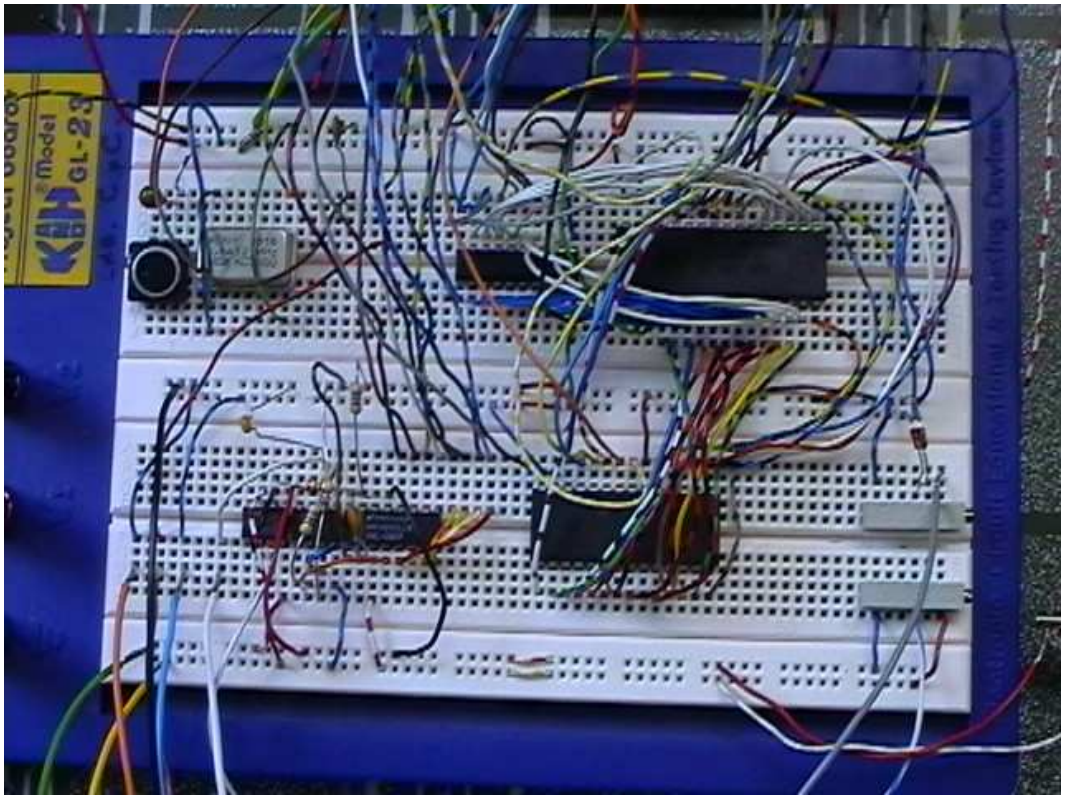


Figura 3.10: Reset, reloj, FlashROM, RAM externa y DAC

para evitar un desbordamiento. Para hacer esto en nuestro código C declararemos dos variables alojadas en los registros de función especial (SFRs):

```
sfr at 0xFF _t3;
sfr at 0x87 _pcon;
```

`_t3` hace referencia al timer del watch dog que, en la documentación del 80592 se indica como *Timer 3*. El otro registro, `_pcon`, es el registro de habilitación de la carga del contador del watch dog. Poniendo a 1 el bit 4, indicamos que vamos a escribir el valor del contador desde software. Para reiniciar el contador del timer y así evitar que se desborde insertaremos regularmente la siguiente secuencia de instrucciones:

```
_pcon = 0x10;
_t3 = 0x00;
```

Lo habitual será insertar esta secuencia dentro del bucle principal del programa o en medio de funciones muy complejas que requieran mucho consumo de ciclos de CPU.

3.8.2 Puertos adicionales

Para acceder a los puertos y capacidades adicionales del 80592 lo único que deberemos hacer en el código es declarar los registro SFR adecuados para trabajar con esas capacidades.

Cojamos como ejemplo el conversor analógico-digital de 10 bits incluido en el procesador. Según la documentación los registros del conversor son el ADCH de sólo lectura situado en la posición C6h y el registro ADCON, de lectura y escritura, situado en la dirección C5h. Para trabajar con el conversor insertaremos, por tanto, las siguientes declaraciones de variables:

```
sfr at 0xC6 _adch;
sfr at 0xC5 _adcon;
```

Para realizar una conversión y una posterior lectura del valor numérico hay que poner en los 3 bits menos significativos del registro ADCON, qué entrada analógica deseamos leer (0 a 7) y, simultáneamente, poner a 1 el bit 3 del mismo registro para iniciar la conversión. Sabremos que la conversión ha terminado cuando el bit 4 del registro ADCON pase de 0 a 1.

Al terminar la conversión es preciso poner a cero todos los bits del registro ADCON para dejar el conversor preparado para una nueva conversión. Si pasamos estos pasos a código nos queda una secuencia muy sencilla:

```
_adcon = 8 | ENTRADA_ANALOGICA;
while ((_adcon & 0x10) == 0)
    ;
_adcon = 0;
valor_convertido = _adch;
```

Así, en `valor_convertido` obtenemos los 8 bits más significativos del valor convertido de la entrada `ENTRADA_ANALOGICA` seleccionada.

De la misma forma se opera con el resto de puertos adicionales que posee el 80592.

3.8.3 Vectores de interrupción adicionales

El procesador 80592 posee 15 vectores de interrupción frente a los 5 vectores del 8051 (ver tabla 2.17).

En la dirección 0x0000 se encuentra el vector de reset. El compilador SDCC, al ser un compilador para el 8051 inicializa sólo los 5 primeros vectores de interrupción con instrucciones `reti`. Cuando queramos crear una rutina de servicio de interrupción tendremos que utilizar el sufijo `interrupt` para esa función seguido de un número que indique de que interrupción se trata (0 a 4). Existe una seria tara en este compilador y es que no permite el uso de vectores de interrupción superiores al 4, con lo cual no podremos dar soporte desde C a ISRs para el ADC, el CAN, etc.

En el apéndice A se detallan las particularidades del compilador SDCC.

3.9 Algunos códigos de ejemplo

Para programar el 80592 en ensamblador existen multitud de ensambladores tanto comerciales como libres, al utilizar el mismo repertorio de instrucciones que el 8051. En este caso, se ha utilizado el ensamblador `asx8051`, que es la versión OpenSource del ensamblador `as51` de Intel.

3.9.1 Conversión analógico_digital

Veamos un código sencillo: un bucle infinito que se encarga de leer en cada iteración el ADC en la entrada 0 y guardar los 8 bits más significativos de la conversión en la variable "var".

```
T3    = 0xFF  ; WDT
PCON  = 0x87  ; Control de WDT
ADCH  = 0xC6  ; Lectura de los 8 bits más significativos del ADC
ADCON = 0xC5  ; Control del ADC

        .area DSEG (DATA,ABS)

        .org    0x30

var:
        .ds     1

        .area CSEG (CODE,ABS)

        .org    0x0000
        ljmp    _main

        .org    0x0003
```

```
        reti                        ; external 0

        .org    0x000B
        reti                        ; timer 0 overflow

        .org    0x0013
        reti                        ; external 1

        .org    0x001B
        reti                        ; timer 1 overflow

        .org    0x0023
        reti                        ; serial i/o (uart)

        .org    0x002B
        reti                        ; serial i/o (can)

        .org    0x0033
        reti                        ; timer 2 capture 0

        .org    0x003B
        reti                        ; timer 2 capture 1

        .org    0x0043
        reti                        ; timer 2 capture 2

        .org    0x004B
        reti                        ; timer 2 capture 3

        .org    0x0053
        reti                        ; adc completion

        .org    0x005B
        reti                        ; timer 2 compare 0

        .org    0x0063
        reti                        ; timer 2 compare 1

        .org    0x006B
        reti                        ; timer 2 compare 2

        .org    0x0073
        reti                        ; timer 2 overflow

_main:
        mov     PCON,#0x10        ; reiniciamos el timer de watch dog
        mov     T3,#0x00
```

```
        mov     ADCON,#0x08       ; leemos la entrada analógica 0
_adc_wait_loop:
        mov     a,ADCON           ; esperamos a que se complete la conversión
        jnb     acc.4,_adc_wait_loop
        mov     ADCON,#0x00
        mov     var,ADCH          ; guardamos el valor convertido en "var"
        sjmp    _main             ; bucle infinito
```

3.9.2 Multiplicación de 16 bits

A continuación un código que calcula una multiplicación de 16 bits, volcando el resultado en 32 bits.

```
        .area DSEG (DATA,ABS)

        .org    0x30

reg_a:
        .ds        4
reg_b:
        .ds        2
acum:
        .ds        4
var1:
        .ds        2
var2:
        .ds        2

        .area CSEG (CODE,ABS)

        .org    0x0000
        ljmp    _main

        .org    0x0003
        reti                        ; external 0

        .org    0x000B
        reti                        ; timer 0 overflow

        .org    0x0013
        reti                        ; external 1
```

```
        .org    0x001B
        reti                        ; timer 1 overflow

        .org    0x0023
        reti                        ; serial i/o (uart)

        .org    0x002B
        reti                        ; serial i/o (can)

        .org    0x0033
        reti                        ; timer 2 capture 0

        .org    0x003B
        reti                        ; timer 2 capture 1

        .org    0x0043
        reti                        ; timer 2 capture 2

        .org    0x004B
        reti                        ; timer 2 capture 3

        .org    0x0053
        reti                        ; adc completion

        .org    0x005B
        reti                        ; timer 2 compare 0

        .org    0x0063
        reti                        ; timer 2 compare 1

        .org    0x006B
        reti                        ; timer 2 compare 2

        .org    0x0073
        reti                        ; timer 2 overflow

_ext_load_16:
        mov       reg_a+0,@r0
        inc       r0
        mov       reg_a+1,@r0
        mov       reg_a+2,#0
        mov       reg_a+3,#0
        ret

_ext_load_32:
        mov       reg_a+0,@r0
        inc       r0
```

```
        mov     reg_a+1,@r0
        inc     r0
        mov     reg_a+2,@r0
        inc     r0
        mov     reg_a+3,@r0
        ret


;-------------------------------------------------------------------------
; r0 y r1 apuntan a dos variables de 16 bits. El resultado de la multiplicacion
; se vuelca en acum (32 bits) (apuntado por r0)
_mul_16:
        acall   _load_16            ; cargar reg_a con variable (r0)
        mov     reg_b,@r1           ; copiar en reg_b el valor de la
        inc     r1                  ; variable apuntada por r1
        mov     reg_b+1,@r1
        acall   _abmul              ; efectuar la multiplicación
        mov     r0,#acum            ; resultado apuntado por r0
        ret
_abmul:
        mov     acum+2,#0           ; parte alta del acumulador a 0
        mov     acum+3,#0
        mov     a,reg_a             ; multiplicar las partes bajas
        mov     b,reg_b             ; (igual a D * B)
        mul     ab
        mov     acum,a              ; acum = resultado provisional
        mov     acum+1,b
        mov     a,reg_a+1           ; parte alta de A por parte baja de B
        mov     b,reg_b             ; (igual a D * A)
        mul     ab
        add     a,acum+1            ; sumar con el resultado parcial
        mov     acum+1,a            ; anterior desplazándolo una
        mov     a,b                 ; posoción a la izquierda
        addc    a,acum+2            ; tener en cuenta un posible
        mov     acum+2,a            ; acarreo hacia acum+2
        jnc     _abm1
        inc     acum+3              ; y hacia acum+3
_abm1:
        mov     a,reg_a             ; multiplicar parte baja de A por
        mov     b,reg_b+1           ; parte alta de B
        mul     ab                  ; (igual a C * B)
        add     a,acum+1
        mov     acum+1,a
        mov     a,b
        addc    a,acum+2
        mov     acum+2,a
        jnc     _abm2
```

```
        inc         acum+3
_abm2:
        mov         a,reg_a+1           ; multiplicar parte alta de A
        mov         b,reg_b+1           ; por parte alta de B
        mul         ab                  ; (igual a A * B)
        add         a,acum+2            ; suma con desplazamiento
        mov         acum+2,a
        mov         a,b
        addc        a,acum+3            ; hay que tener en cuenta los
        mov         acum+3,a            ; posibles carreos
        ret

_main:
        mov         var1,#0x20          ; var1 = 0x0020
        mov         var1+1,#0x00
        mov         var2,#0x14          ; var2 = 0x0114
        mov         var2+1,#0x01
        mov         r0,#var1
        mov         r1,#var2
        lcall     _mul_16             ; multiplicamos var1 * var2
        ljmp      .                   ; bucle infinito
```

Capítulo 4

El emulador de FlashROM

Para el desarrollo de software en el prototipo es necesario programar la memoria FlashROM mediante un programador adecuado cada vez que se quiera probar un código. Este es un método a todas luces enrevesado, porque si bien para grabar software de explotación es la única solución posible, se trata de todo un inconveniente cuando de lo que se trata es de probar y depurar software; principalmente porque un simple error en el código fuente puede costarnos varios minutos de tiempo al tener que reprogramar de nuevo la totalidad de la Flash.

La solución adoptada finalmente fue la construcción de un emulador hardware de memorias ROM. Dicho emulador se conecta al PC a través del puerto paralelo y permite emular, mediante una memoria RAM alimentada de forma independiente, una memoria ROM, sustituyendo, de esta forma, la FlashROM 29F010 directamente en el zócalo del prototipo. Ver la figura 4.1.

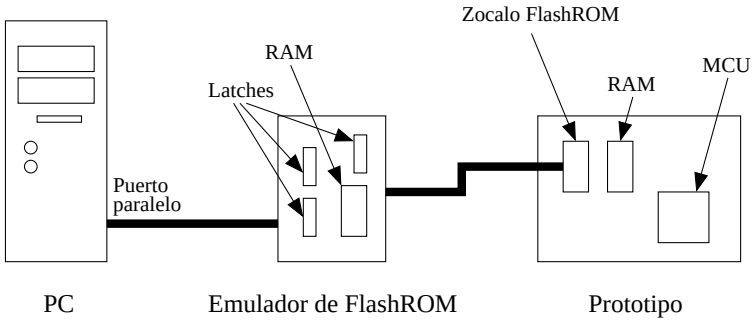


Figura 4.1: Esquema del prototipo con el emulador

De esta manera nos es posible probar el software antes de ponerlo en explotación

4.1 Implementación del emulador de FlashROM

El emulador consta de unos circuitos de latch montados alrededor de una pastilla RAM 62256 de 32 Kbytes. Los latches L1 y L2 se encargan de cargar la parte alta y la parte baja del bus de direcciones, respectivamente, desde el PC. El latch L3 permite hacer lo mismo con las señales del bus de control de la RAM (ver figura 4.2).

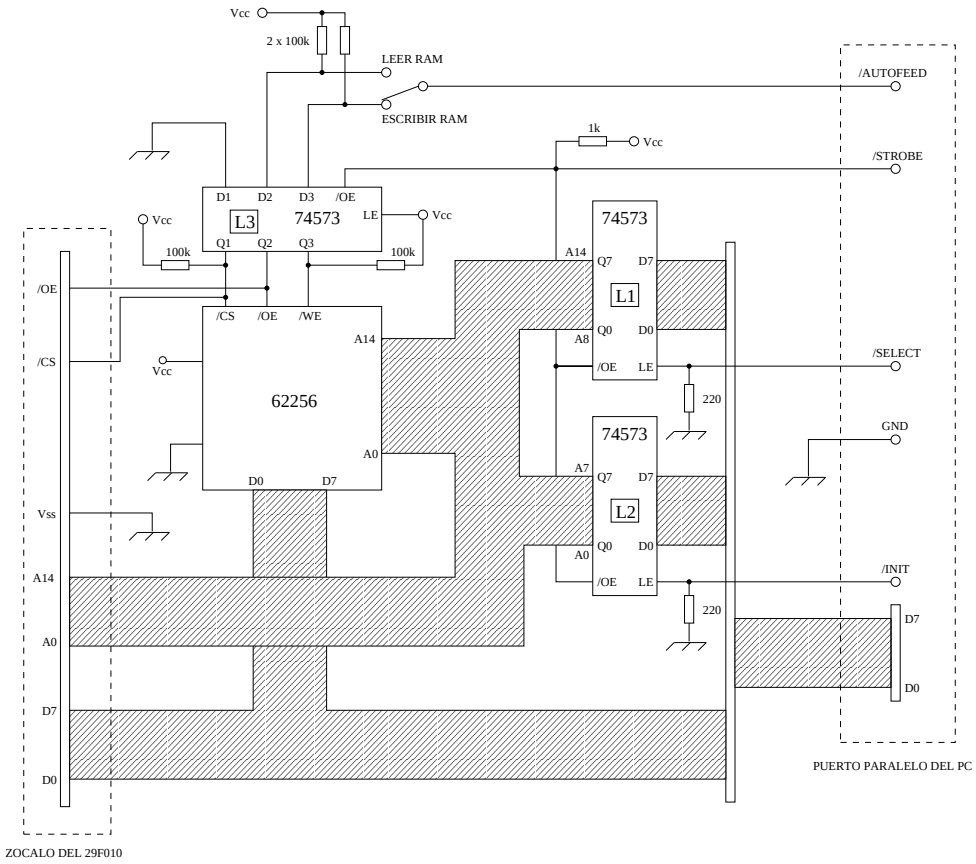


Figura 4.2: Emulador FlashROM

Mediante la señal /STROBE desde el PC podemos poner los 3 latches en alta impedancia (a 1) o en modo activo (a 0). Pondremos los latches en alta impedancia cuando queramos "desconectar"la pastilla de RAM del PC y que pueda conectarse al prototipo manteniendo los datos en su interior. Los pondremos en modo activo cuando vayamos a leer o escribir en la RAM desde el PC.

Cuando los latches están activos, utilizamos las señales /SELECT para cargar la parte alta del bus de direcciones e /INIT para cargar la parte baja, y luego utilizamos la señal /AUTOFEED para indicar si queremos hacer una lectura o una escritura en esa posición de la RAM. El interruptor de LECTURA/ESCRITURA permite encauzar la señal /AUTOFEED del puerto paralelo a las entradas de control /OE (en la posición de lectura) o /WE (en la posición de escritura). Debido a las limitaciones propias de los puertos paralelos en un PC, el modo de escritura o lectura se pone de forma manual, mediante un interruptor etiquetado. Esto es así debido a que el puerto paralelo carece de más salidas de control (ver tabla 4.1).

E/S	Pin DB25	Pin Centronics	Nombre	Bit	Utilización
S	1	1	/strobe	C0-	Habilitar latches
S	2	2	data 0	D0	Bit menos significativo
S	3	3	data 1	D1	.
S	4	4	data 2	D2	.
S	5	5	data 3	D3	.
S	6	6	data 4	D4	.
S	7	7	data 5	D5	.
S	8	8	data 6	D6	.
S	9	9	data 7	D7	Bit más significativo
E	10	10	/ack	S6+	
E	11	11	/busy	S7-	
E	12	12	/pe	S5+	
E	13	13	/slctin	S4+	
S	14	14	/autofd	C1-	Orden de lectura o escritura
E	15	32	/error	S3+	
S	16	31	/init	C2+	Parte baja de la dirección
S	17	36	/select	C3-	Parte alta de la dirección
	18-25	19-30	ground		

Tabla 4.1: Pines del puerto paralelo

Las direcciones base de los puertos paralelo son: 0x3BC (para LPT0), 0x378 (LPT1) y 0x278 (para LPT2). En todos los PCs el puerto paralelo incluido por defecto es el LPT1 alojado en 0x378. El control y acceso al puerto paralelo se realiza mediante los 3 puertos de entrada/salida dispuestos en las direcciones BASE, BASE+1 y BASE+2 (siendo BASE=0x3BC, 0x378 o 0x278, según el caso). Ver las tablas 4.2, 4.4 y **??** con el mapeado de los bits del puerto paralelo en los puertos de entrada salida del PC.

D7	D6	D5	D4	D3	D2	D1	D0
MSB							LSB

Tabla 4.2: Datos (dirección BASE). Lectura/Escritura.

S7-	S6	S5	S4	S3	S2	S1	S0
/busy	/ack	/pe	/slct in	/error	-	-	-

Tabla 4.3: Estado (Dirección BASE + 1). Sólo lectura.

La nomenclatura seguida es muy sencilla. Cada bit está señalizado mediante una letra en mayúscula que indica el tipo de bit (D = dato, S = estado (sólo lectura) y C = control (sólo escritura)), seguida de un dígito entre el 0 y el 7 que indica el bit dentro del byte del puerto de entrada/salida y de un signo menos opcional. Dicho signo menos indica que el valor de ese bit en el puerto de entrada/salida está invertido con respecto al valor del bit en el conector del puerto paralelo del PC (Por ejemplo, si pusiésemos a 0 el bit

C7	C6	C5	C4	C3-	C2	C1-	C0-
-	-	hz	irq en	/select	/init	/autofd	/strobe

Tabla 4.4: Control (Dirección BASE + 2). Sólo escritura.

1 del puerto de entrada/salida BASE + 2 (control), que está etiquetado como "C1-", el pin correspondiente del conector del PC se pondría a 1). Los bits que no posean este signo menos opcional se comportarán de la manera usual (un 0 provoca/lee un nivel bajo, mientras que un 1 provoca/lee un nivel alto).

El bit C4 no se utiliza en nuestro sistema. El bit C5 (hz) se utiliza para habilitar el puerto paralelo como puerto de entrada de datos. Esta capacidad sólo es accesible en ordenadores equipados con un puerto paralelo compatible PS/2 (formalemente se trata de un EPP, *Enhanced Parallel Port*, según el estándar de IBM). Con este bit a 0 el puerto funciona de la manera usual, como puerto de salida de datos, siendo el puerto de entrada/salida BASE de sólo escritura. Si ponemos el bit C5 a 1, los pines de puerto paralelo D0 a D7 pasan a ser bits de entrada y el puerto de entrada/salida BASE pasa a ser de sólo lectura. El estándar EPP indica, a su vez, que cuando ponemos el puerto paralelo en modo lectura, los pines D0 a D7 pasan a un modo de lectura en alta impedancia, con lo que, al mismo tiempo que se lee, se realiza una desconexión eléctrica de esos pines con el exterior.

Es esta capacidad de bidirección en el puerto paralelo la que nos permitirá realizar operaciones de lectura sobre el emulador desde el PC para verificar la integridad de lo que escribamos.

De esta forma resumimos de forma sencilla lo que serían los 3 modos de funcionamiento del emulador ROM:

- *Modo lectura*: Todos los latches activos y el interruptor manual en la posición de LECTURA. Se pone el puerto paralelo en modo normal (de salida) y se carga la dirección de memoria en los latches L1 y L2. A continuación se pone el puerto paralelo en modo de entrada y enviamos la orden a través de la línea /AUTOFEED para leer un dato de la RAM. De esta forma el dato direccionado se puede leer de la dirección base del puerto paralelo.

- *Modo escritura*: Todos los latches activos y el interruptor manual en la posición de ESCRITURA. Se pone el puerto paralelo en modo normal (de salida) y se carga la dirección de memoria en los latches L1 y L2. A continuación, manteniendo el puerto en modo normal, situamos el byte que se quiere escribir en el puerto y activamos (a 0) la línea /AUTOFEED para indicarle a la RAM que cargue el dato que le damos.

- *Modo de alta impedancia*: Todos los latches en modo de alta impedancia (/STROBE a 1). De esta forma, la RAM queda virtualmente desconectada del PC salvo por su bus de datos. Pero, ¿Qué sucede con el bus de datos? La especificación oficial de IBM para el puerto puerto paralelo bidireccional de su modelo PS/2 (el estándar de

facto que se utiliza en todos los equipos PC actuales) indica claramente que cuando el puerto paralelo se pone en modo de entrada, este modo es, a la vez, un modo de alta impedancia. Con lo cual tenemos que, es posible aislar totalmente el PC del emulador, situando la señal /STROBE a 1 y poniendo el puerto paralelo en modo de entrada (alta impedancia).

Cuando se implementó este emulador en el laboratorio surgió el problema de que la implementación del puerto paralelo PS/2 en algunos fabricantes no se respeta, dándose el caso de que, cuando se ponía en modo de entrada no se ponía en modo de alta impedancia (!). En caso de que nos encontremos con puertos como éstos que no respetan el estándar la mejor opción es desconectar físicamente el emulador del PC, para ello se ha habilitado la resistencia en pull-up de 1K situada en la línea de /STROBE que permite poner el emulador en el modo de alta impedancia por defecto.

En la figura 4.3 podemos ver una fotografía del emulador de FlashROM implementado.

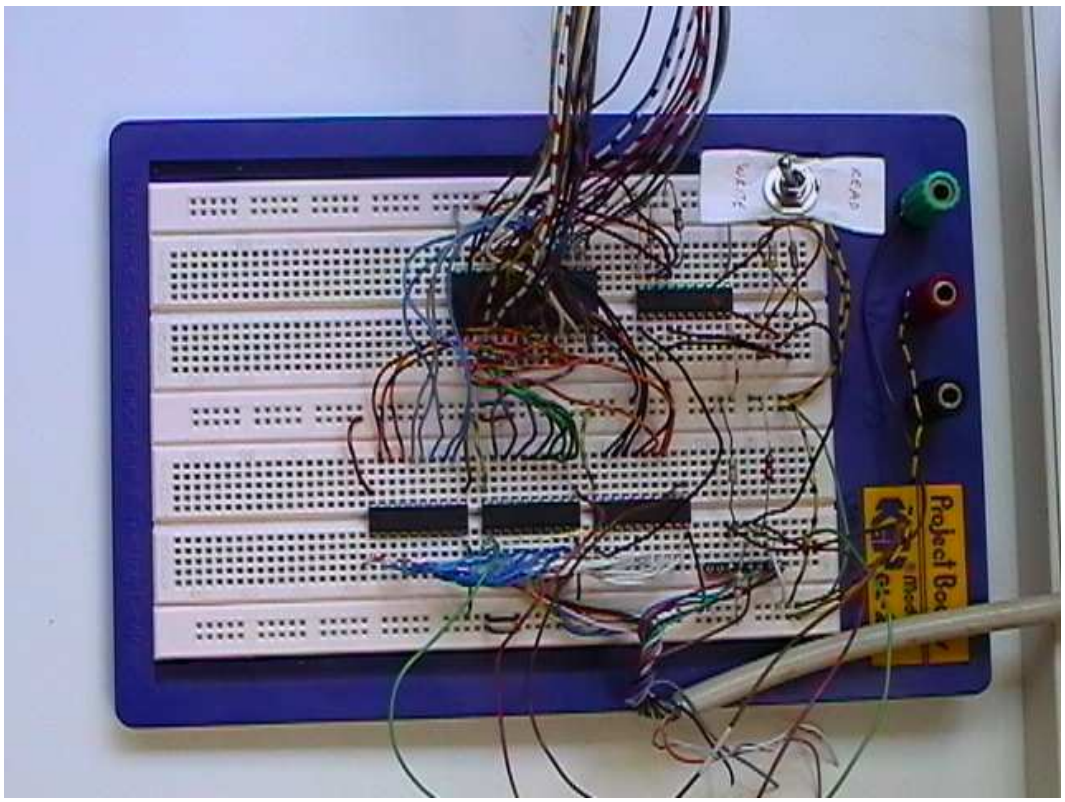


Figura 4.3: El emulador de FlashROM

4.2 La API del emulador

Para acceder al emulador de FlashROM se han creado dos ficheros `flashemu.h` y `flashemu.c`, que permiten acceder al emulador mediante sencillas llamadas en C. A continuación se

encuentra listado el fichero `flashemu.c` en el que se puede apreciar claramente cómo se manejan los bits para realizar lecturas y escrituras en el emulador de FlashROM.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <asm/io.h>

#define  LP_BASE     0x378
#define  DELAY_TIME  20


void flash_emu_init(void) {
    ioperm(LP_BASE, 3, 1);    /* permitimos el acceso a los puertos */
    outb(0, LP_BASE);         /* ponemos un 0 en los datos */
    outb(0x09, LP_BASE + 2);  /* ponemos el /OE de los 3 latches a 0 */
    return;
    }

void flash_emu_send_data(unsigned char data) {
    outb(data, LP_BASE);
    return;
    }

void flash_emu_enable_high_address(void) {
    /* enviamo un 1 al pin LE del latch L1 */
    outb(inb(LP_BASE + 2) & 0xF7, LP_BASE + 2);
    usleep(DELAY_TIME);
    /* enviamos un 0 al pin LE del latch L1 */
    outb(inb(LP_BASE + 2) | 0x08, LP_BASE + 2);
    return;
    }

void flash_emu_enable_low_address(void) {
    /* enviamos un 1 al pin LE del latch L2 */
    outb(inb(LP_BASE + 2) | 0x04, LP_BASE + 2);
    usleep(DELAY_TIME);
    /* enviamos un 0 al pin LE del latch L2 */
    outb(inb(LP_BASE + 2) & 0xFB, LP_BASE + 2);
    return;
    }

void flash_emu_enable_data(void) {
    /* enviamos un 0 a través de /AUTOFD para habilitar la lectura/escritura */
    outb(inb(LP_BASE + 2) | 0x02, LP_BASE + 2);
```

```
    usleep(DELAY_TIME);
    /* enviamos un 1 a través de /AUTOFD para retornar al estado de reposo */
    outb(inb(LP_BASE + 2) & 0xFD, LP_BASE + 2);
    return;
    }

void flash_emu_write_at(unsigned int address, unsigned char data) {
    /* enviamos los 8 bits más significativos de la dirección */
    flash_emu_send_data(address >> 8);
    usleep(DELAY_TIME);
    /* cargamos el latch L1 con esos 8 bits */
    flash_emu_enable_high_address();
    usleep(DELAY_TIME);

    /* enviamos los 8 bits menos significativos de la dirección */
    flash_emu_send_data(address & 0x00FF);
    usleep(DELAY_TIME);
    /* cargamos el latch L2 con esos 8 bits */
    flash_emu_enable_low_address();
    usleep(DELAY_TIME);

    /* enviamos los 8 bits del dato */
    flash_emu_send_data(data);
    usleep(DELAY_TIME);
    /* cargamos la RAM con esos 8 bits */
    flash_emu_enable_data();
    usleep(DELAY_TIME);
    return;
    }

unsigned char flash_emu_read_at(unsigned int address) {
    unsigned char ret;

    /* enviamos los 8 bits más significativos de la dirección */
    flash_emu_send_data(address >> 8);
    usleep(DELAY_TIME);
    /* cargamos el latch L1 con esos 8 bits */
    flash_emu_enable_high_address();
    usleep(DELAY_TIME);

    /* enviamos los 8 bits menos significativos de la dirección */
    flash_emu_send_data(address & 0x00FF);
    usleep(DELAY_TIME);
    /* cargamos el latch L2 con esos 8 bits */
    flash_emu_enable_low_address();
    usleep(DELAY_TIME);
```

```
    /* ponemos el puerto paralelo en modo de entrada/alta impedancia */
    outb(0x29, LP_BASE + 2);
    /* damos la orden de lectura a través del pin /AUTOFD */
    outb(inb(LP_BASE + 2) | 0x02, LP_BASE + 2);
    /* leemos el dato de 8 bits */
    ret = inb(LP_BASE);
    /* ponemos el puerto paralelo de nuevo en modo de salida */
    outb(inb(LP_BASE + 2) & 0xFD, LP_BASE + 2);
    outb(0x09, LP_BASE + 2);
    return ret;
    }

void flash_emu_done(void) {
    /* ponemos a 1 las entradas /OE de los 3 latches y ponemos el puerto
       paralelo en modo de entrada/alta impedancia para desconectar el PC de la
       pastilla RAM. */
    outb(0x20, LP_BASE + 2);
    ioperm(LP_BASE, 3, 0);
    return;
    }
```

La función `flash_emu_write_at` escribe un dato de 8 bits en una dirección de la RAM del emulador y, para que funcione, el conmutador del emulador debe estar en la posición "ESCRIBIR RAM". La función `flash_emu_read_at` lee un byte de una dirección de la RAM del emulador y para que funcione, el conmutador del circuito debe estar en la posición de "LEER RAM".

4.3 El software para el emulador

Se ha creado una aplicación sencilla por línea de comandos que permite leer ficheros HEX y volcarlos al emulador FlashROM. Dicha aplicación, llamada `ihxwrite`, lee un fichero Intel HEX y lo vuelca a través del puerto paralelo al emulador FlashROM. La aplicación accede directamente a los puertos del PC por lo que se trata de un programa no portable a otras plataformas, aunque está programado en C para Linux. En la sección B.1 se encuentra el listado de este programa.

La sintaxis de la línea de comandos para el programa `ihxwrite` es:

```
ihxwrite fichero.ihx [-nr]
```

La opción `-nr` indica al programa que no realice un test para verificar la integridad de la RAM del emulador. Al invocar el programa `ihxwrite`, éste nos pedirá que pongamos el conmutador del emulador en la posición "ESCRIBIR RAM"y que pulsemos Intro. Tras

esto, el programa irá grabando la RAM del emulador con el contenido del fichero Intel HEX. Al finalizar la escritura de los datos en el emulador, si no hemos puesto la opción `-nr` por línea de comandos, el programa nos pedirá que pongamos el conmutador del emulador en modo "LEER RAM"y, tras pulsar intro, procederá a la lectura de los datos del emulador para verificar que se han escrito bien. Si indicamos la opción `-nr`, tras la fase de escritura el programa termina la ejecución.

Además del programa `ihxwrite`, se ha implementado un programa auxiliar: `ihxtest`, cuya sintaxis es `ihxtest fichero.ihx` y que simplemente compara el contenido de la RAM del emulador con el fichero Intel HEX indicado por línea de comandos. Este programa sólo accede al emulador en modo lectura (antes de proceder a la lectura de los datos, el software indica al usuario que ponga el conmutador del emulador en la posición "LEER RAM"). El código de este software se encuentra en la sección B.2.

Para implementar un programa de aplicación en la placa 80592 y ejecutarlo el procedimiento es ahora sencillo:

- Generamos el fichero Intel HEX a partir del código fuente mediante el compilador:

  ```
  sdcc ejemplo.c
  ```

  Esta acción genera, entre otros, el fichero `ejemplo.ihx` con el programa para 8051 en código máquina.

- Volcamos el fichero `ejemplo.ihx` en el emulador de FlashROM ó, mediante un programador de EEPROM lo grabamos en una Flash. En el primer caso haremos un `ihxwrite ejemplo.ihx` con el emulador conectado al puerto paralelo y alimentado (el software nos irá dando instrucciones para cambiar el interruptor LECTURA/ESCRITURA cuando sea necesario).

- Con el emulador de FlashROM desconectado del PC y conectado a la placa del 80592, o con la memoria Flash ya grabada insertada en la placa del 80592, alimentamos dicha placa y comprobaremos que el micro está corriendo nuestro programa, escrito inicialmente en C.

De esta forma se *cierra el círculo* y tenemos el sistema de desarrollo preparado para implementar aplicaciones en él.

Capítulo 5

Microcyc Fuzz : Aplicación de control borroso

Como ejemplo de aplicación en el prototipo se ha implementado un control borroso. Para implementar el control borroso se ha utilizado el compilador SDCC junto con un preprocesador de lenguaje de alto nivel. El resultado es un sistema fácilmente mantenible y cuya eficiencia depende en mayor medida de la eficiencia del compilador para optimizar el código C.

5.1 El software

Para la implementación de un software en el microcontrolador para control borroso existen básicamente, dos posibilidades.

- Crear una librería sencilla que se linke con nuestros programas para el SDCC y que permite definir conjuntos difusos, reglas, etc.
- Crear el software exactamente a la medida para cada aplicación concreta.

En el primer caso obtenemos un software muy fácil de mantener pero poco eficiente y con tendencia a generar códigos grandes, mientras que en segundo caso tenemos un código pequeño y eficiente pero muy difícil de mantener y de reutilizar.

La medida adoptada finalmente fue crear un traductor de lenguaje, que denominamos *fuzz*, a lenguaje C. El concepto es sencillo: nos inventamos un lenguaje de muy alto nivel que permite crear conjuntos y variables difusas así como definir reglas de control difuso sobre las variables; y nuestro software lo que hará será traducir dicho código fuente en código C exactamente a la medida de la aplicación. De esta manera tenemos un código

fácil de mantener y sencillo (escrito en lenguaje `fuzz`) y, a la vez, un código C eficiente, mediante el traductor.

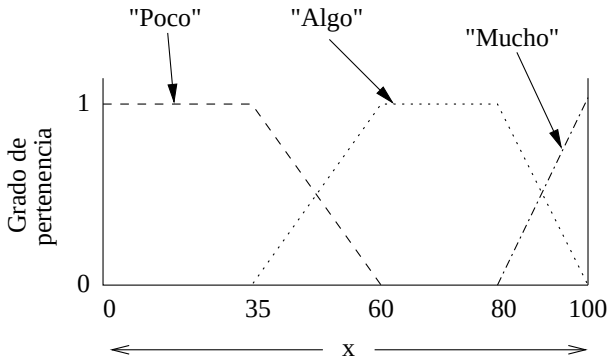
Veamos un ejemplo de código `fuzz`:

```
set bajo  = {0, 0, 0, 50}
set medio = {0, 50, 50, 100}
set alto  = {0, 100, 100, 100}
set poco  = {0, 0, 0, 100}
set mucho = {0, 100, 100, 100}

var entrada at adc 3 = bajo medio alto
var salida1 at 0x27 = poco mucho
var salida2 at 0x28 = poco mucho

if (entrada is bajo) or (entrada isnot medio) then
    salida2 is poco
if (entrada is alto) and (entrada is medio) then
    salida1 is poco
    salida2 is mucho
```

Mediante la cláusula `set` definimos los conjuntos difusos. Un conjunto difuso lo definiremos como un trapezoide, suponiendo que siempre utilizamos conjuntos difusos normalizados. Cada uno de los cuatro valores encerrados entre llaves indica un punto en el eje x (ver la figura 5.1). Los valores en el eje x están entre 0 y 100 (ambos inclusive) indicando valores mínimo y máximo para una variable de tipo byte en el 80592 (el 0 del conjunto difuso corresponde al 0 de la variable y el 100 del conjunto difuso corresponde al 255 de la variable).




```
set poco = {0, 0, 35, 60}
set algo = {35, 60, 80, 100}
set mucho = {80, 100, 100, 100}
```

Figura 5.1: Ejemplo de cómo se definen los conjuntos difusos

La palabra reservada `var` permite definir variables difusas y asociarles los conjuntos difusos que queramos. Mediante la cláusula `at` indicamos de donde vamos a leer esa variable físicamente o dónde escribiremos su valor físicamente, dándose tres casos:

- `at número`. Para indicar un registro de función especial en el 80592. Estas variables pueden ser tanto de entrada como de salida.
- `at adc número`. Para indicar una entrada analógica del micro. En este caso las variables sólo podrán ser de entrada.
- `at anywhere`. Para indicar que la variable borrosa que estamos declarando estará asociada a una variable `unsigned char` de C estándar.

En el ejemplo anterior tenemos definida una variable a la que llamamos `entrada` que tomamos de la entrada analógica número 3 del 80592 y dos variables más a las que llamamos `salida1` y `salida2` y que alojamos en las posiciones de memoria 0x27 y 0x28 respectivamente.

La cláusula `at anywhere` aunque en un principio pueda resultar de poca utilidad ya que no se asocia a ninguna entrada o salida, en un SFR o en el ADC, es útil cuando deseamos retocar el código C resultante del procesado y aplicar algún tratamiento numérico al valor de alguna entrada (antes de fuzzificar) o salida (después de defuzzificar). En la sección 5.3 se observa una aplicación de este tipo de variables.

Después de definir los conjuntos y la variables difusas, definimos las reglas que se utilizarán.

```
if expresion then
    lista_de_sentencias
```

Las expresiones que se utilizan son las del tipo del ejemplo, expresiones booleanas sobre operaciones de pertenencia. Las operaciones booleanas permitidas son las usuales de `op and op`, `op or op` ó `not op`; siendo `op` una operación de pertenencia del tipo `variable is conjunto` ó `variable isnot conjunto`. Por ejemplo:

```
if (var1 is conj1) and ((var2 is conj2) or (var3 isnot conj4)) then
    var3 is conj6
    var6 is conj2
```

La precedencia de operadores es la usual de cualquier lenguaje, permitiéndose el uso de paréntesis en la expresión de evaluación del `if`. No pueden haber variables definidas como `at adc ...` después del `then`, ya que son siempre de salida. Nótese, además, que si tenemos una evaluación de pertenencia del tipo `v is c`, el conjunto difuso `c` tiene que haber sido incluido como conjunto difuso para la variable `v` mediante la cláusula `var` correspondiente, por ejemplo:

```
var v at 0x37 = a b c
```

El método de defuzzificación utilizado es el de la media de centros, al ser sencillo de implementar y muy rápido.

El software, denominado `microcyc_fuzz` acepta dos parámetros, el primero es el fichero fuente escrito en lenguaje `fuzz` y el segundo parámetro es el nombre del fichero .c donde el programa escribirá el código de salida.

```
./microcyc_fuzz ejemplo.fuzz ejemplo.c
```

Una vez hecha esta operación, en el fichero `ejemplo.c` tendremos el código C ya traducido y preparado para ser compilado y que se ejecute en el microcontrolador. Si echamos un vistazo a este código veremos que, entre otras funciones, posee la función `fuzz_init` y la función `fuzz_work`. La función `fuzz_init` se encarga de inicializar las variables difusas a los conjuntos y la función `fuzz_work` se encarga de iterar un vez las reglas que hemos definido. Ambas funciones son hechas a la medida de la aplicación consiguiendo de esta forma un funcionamiento más rápido.

La función `main` que se define es muy sencilla:

```
void main(void) {
    fuzz_init();
    while (1) {
        fuzz_work();
        }
    }
```

De esta forma el microcontrolador, al reiniciar inicializa las variables difusas y entra en un bucle infinito en el que itera una y otra vez las reglas de control borroso que hemos implementado[1].

Nada impide añadir código adicional al fichero de salida. Dicho código se compilará de forma usual, mediante `sdcc ejemplo.c` y el código objeto resultante `ihx` se ejecutará de la forma que queramos en el procesador (ya sea mediante el emulador de FlashROM o tostando físicamente una ROM).

5.1.1 La gramática del lenguaje *fuzz*

A continuación se incluye un extracto del código del programa YACC (en este caso `bison` para ser más exactos) en el que se aprecia cómo está organizada la gramática de lenguaje

[1] La función `main` en la realidad incluye, además, código para evitar el desboramiento del timer del watch dog en cada iteración

```
%token ID
%token LEFTPAR
%token RIGHTPAR
%token IF
%token THEN
%token AND
%token OR
%token NOT
%token IS
%token ISNOT
%token SET
%token VAR
%token EQUAL
%token ADC
%token AT
%token ANYWHERE
%token CONST

%left OR
%left AND
%left NOT

%%

program             : setList varList ruleList
                    ;

setList             : setList setDefinition
                    | setDefinition
                    ;

setDefinition       : SET ID EQUAL CONST CONST CONST CONST
                    ;

varList             : varList varDefinition
                    | varDefinition
                    ;

varDefinition       : VAR ID AT where EQUAL setIdList
                    ;

where               : CONST
                    | ADC CONST
```

```
                    | ANYWHERE
                    ;

setIdList           : setIdList ID
                    | ID
                    ;

ruleList            : ruleList ruleDefinition
                    | ruleDefinition
                    ;

ruleDefinition      : IF fact THEN resultList
                    ;

fact                : fact AND fact
                    | fact OR fact
                    | NOT fact
                    | LEFTPAR fact RIGHTPAR
                    | uFact
                    ;

uFact               : ID varRelation ID
                    ;

varRelation         : IS
                    | ISNOT
                    ;

resultList          : resultList result
                    | result
                    ;

result              : ID IS ID
                    ;

%%
```

5.2 El ejemplo de aplicación

Como ejemplo de aplicación del sistema de control borroso, se ha utilizado una de las plantas existentes en el laboratorio de Computadoras y Control del departamento: un prototipo a escala de un puente de grúa. El prototipo consta de un vagón que se mueve sobre una guía de 2 railes y que transporta un contenedor suspendido. El movimiento del vagón se consigue actuando sobre 2 servomotores mediante la aplicación de un voltaje.

Así, si se desea parar el vagón, se aplicarán 0 voltios; si se desea avanzar hacia la izquierda, se aplicarán voltajes negativos (a mayor valor absoluto del voltaje, mayor velocidad); mientras que si se desea avanzar hacia la derecha, se aplicarán voltajes positivos (a mayor valor absoluto del voltaje, mayor velocidad). En la figura 5.2 se muestra de forma esquemática la disposición física de la grúa.

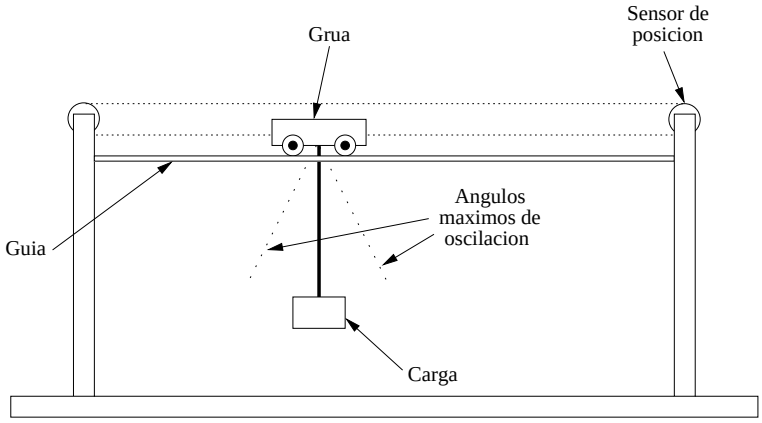


Figura 5.2: Esquema de la grúa con la carga colgante

El vagón posee, además una pequeña carga colgante, con un potenciómetro en el extremo del vagón que permite determinar el ángulo de inclinación de la carga con respecto a la vertical.

En nuestro caso el objetivo del control es el de mover la grúa a un punto de la guía manteniendo en todo momento la carga lo más vertical posible (evitando las oscilaciones de la carga). El punto de la guía elegido como punto de destino, por comodidad, es el punto central del recorrido de la grúa.

De esta manera, si situamos en un momento inicial la grúa en el extremo izquierdo de la guía, se desplazará hacia la derecha hasta alcanzar la consigna y, de la misma manera, si ponemos en un momento inicial la grúa en el extremo derecho de la guía, el vehículo se desplazará hacia la izquierda hasta alcanzar la consigna (la posición central). Todo esto, manteniendo la carga colgante lo más vertical posible.

5.2.1 Conexionado de la grúa con el Microcyc

El vagón posee un conector de 5 pines con las entradas de alimentación, la entrada para control del motor de avance y la salida para leer el ángulo de la carga con respecto a la vertical. Además, para medir la posición del vehículo, en el extremo derecho de la guía horizontal se encuentra un potenciómetro que actúa como divisor de tensión, permitiendo, mediante un voltaje, medir la posición de la grúa.

El conector de la grúa es el siguiente:

Blanco	Gris	Verde	Amarillo	Negro
Motor	Ángulo	+12 V	-12 V	Masa

El cable etiquetado como "Motor"es la entrada de control del motor hacia la grúa. El valor de esta entrada puede oscilar entre -7.5 V (máxima velocidad hacia la izquierda) y +7.5 V (máxima velocidad hacia la derecha). El cable etiquetado como "Ángulo"emite un voltaje proporcional al álgulo de la carga sobre la vertical. La vertical da como salida un voltaje de 2.5 V, una desviación hacia la derecha emite voltajes inferiores, mientras que una desviación hacia la izquierda da como resultado voltajes mayores. El resto de cables son los de alimentación.

Los cables del potenciómetro sensor de posición tienen la siguiente asignación:

Marrón	Azul	Naranja
Pos	+5 V	Masa

El sensor de posición está configurado como un partidor de tensión mediante un potenciómetro multivuelta. Al conectar los cables azul y naranja a +5 y 0 voltios respectivamente, obtenemos en el cable marrón un voltaje entre 0 y 5 voltios proporcional a la posición de la grúa. Las posiciones a la derecha provocan voltajes mayores que las posiciones a la izquierda.

El recorrido de la grúa a través de la guía no coincide con el recorrido total del potenciómetro multivuelta. Si, por ejemplo, calibramos el potenciómetro para que la posición del vehículo más a la izquierda sea 0 voltios en el cable marrón, la posición más a la derecha no llegará a dar una medida de 5 voltios. La solución adoptada fue centrar ambos recorridos: se coloca el vagón a la mitad del recorrido y se calibra el potenciómetro para que el cable de posición de un valor de 2.5 voltios (la mitad del recorrido del potenciómetro multivuelta).

Lectura del sensor de posición

El sensor de posición es el sensor de más fácil lectura. Al estar el conversor analógico-digital del 80592 calibrado para medir voltajes de entre 0 y 5 voltios, la conexión del cable marrón del potenciómetro multivuelta con el microcontrolador es directa. En nuestro caso hemos utilizado la entrada analógica 2 del chip (aunque se pudo haber utilizado cualquier otra).

Lectura del sensor del ángulo

El sensor del ángulo, situado en el vehículo, cuando se encuentra en la vertical produce un voltaje de 4.4 voltios. Al desviarse hacia la derecha emite voltajes menores y al desviarse

hacia la izquierda, voltajes mayores. Estos voltajes están limitados a aproximadamente 0 y 8 voltios respectivamente, siendo realmente el rango de salida de -12 a +12 voltios.

Aunque, como se ha visto, el rango de voltaje que emite el sensor del ángulo es de -12 a +12 voltios, en la práctica las medidas de ángulo que se realizan están en el rango de 0 a 8 voltios (debido al peso de la carga y a la velocidad máxima a la que puede moverse el vehículo).

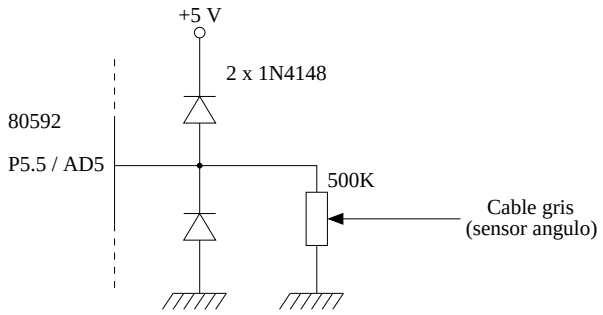


Figura 5.3: Circuito para acondicionar la señal proveniente del sensor del ángulo de la carga

Conectando un partidor de tensión seguido de un limitador de voltaje construido mediante diodos (ver figura 5.3) podemos realizar lecturas del ángulo de la carga en el rango del conversor analógico-digital (entre 0 y 5 voltios), ajustando el partidor de tensión con la carga en la vertical hasta que tengamos un voltaje de entrada en el pin del microcontrolador de 2.5 voltios. Ajustando el circuito de esta forma, obtenemos valores menores de 127 en el ADC para ángulos de desviación hacia la derecha y valores mayores de 127 para ángulos de desviación hacia la izquierda.

La entrada analógica utilizada para leer el sensor del ángulo es la 5, como se puede ver en la figura anterior.

Actuación sobre la velocidad del motor

Para actuar sobre la velocidad del motor utilizaremos el DAC externo de 8 bits alojado a la salida del puerto P4 del microcontrolador (ver la sección 3.7). De esta manera, enviando un valor numérico al puerto P4, generamos un voltaje proporcional en la salida del DAC.

El cable que permite controlar la velocidad del motor es el blanco y el rango de tensión de control es de -7.5 (máxima velocidad hacia la izquierda) a +7.5 voltios (máxima velocidad hacia la derecha). Con el DAC montado lo único que hay que hacer es regular el potenciómetro de fondo de escala de tal manera que la configuración binaria 00000000 genere un voltaje de -7.5 voltios, y la configuración binaria 11111111 genere un voltaje de +7.5 voltios.

Ya tenemos preparado el hardware para realizar la aplicación del control borroso sobre la planta.

5.2.2 El código *fuzz* para el control de la planta

A continuación se puede ver el código fuente, en lenguaje *fuzz* para el control de la planta.

```
set bajo  = 0 0 40 50
set medio = 40 50 50 60
set alto  = 50 60 100 100

var posicion   at adc 2 = bajo medio alto
var angulo     at adc 5 = bajo medio alto
var movimiento at 0xC0  = bajo medio alto


/*  Control del movimiento en función de la posición. */
if posicion is alto then
    movimiento is bajo
if posicion is bajo then
    movimiento is alto
if posicion is medio then
    movimiento is medio

/*  Control del movimiento en función del ángulo de la carga. */
if angulo is alto then
    movimiento is bajo
if angulo is bajo then
    movimiento is alto
```

Definimos tres conjuntos difusos: `bajo`, `medio` y `alto`; y las tres variables difusas implicadas en el control:

```
    var posicion   at adc 2 = bajo medio alto
    var angulo     at adc 5 = bajo medio alto
    var movimiento at 0xC0  = bajo medio alto
```

Las tres variables las definimos sobre los mismos conjuntos difusos (se podrían utilizar otros cualesquiera). La variable `posicion` la asociamos a la entrada analógica 2, la variable `angulo` la asociamos a la entrada analógica 5, y la variable `movimiento` la asociamos al registro de función especial 0xC0, que es el registro asociado al puerto P4 (puerto al que está conectado el DAC externo). Las dos primeras, al ser variables "adc" son variables de entrada, mientras que la tercera la utilizamos como variable de salida.

La reglas de control borroso están divididas en dos grupos:

- Control del motor en función de la posición.
- Control del motor en función del ángulo de la carga.

En el primer grupo de reglas (las tres primeras) intentamos que el vagón grúa se dirija al centro del conjunto difuso etiquetado como `medio` y que corresponde con el punto medio de la guía. En el segundo grupo de reglas tratamos de contrarrestar los movimientos oscilatorios de la carga (variable de entrada `angulo`) actuando sobre el movimiento (si el ángulo de la carga se desplaza hacia la derecha, tratamos de desplazar el vagón hacia la derecha, y si el ángulo de la carga se desplaza hacia la izquierda tratamos de desplazar el vehículo hacia la izquierda).

Nótese que no se está realizando el control de la forma usual (a partir de la señal error); sino que el procesador lee directamente las salidas de la planta (posición y ángulo) y en función de éstas, actúa sobre la entrada de la planta (el voltaje del motor). Este método, aunque poco ortodoxo, permite una mayor simplicidad a la hora de describir el control implementado (en la sección 5.3 se puede ver la forma en la que se utiliza *fuzz* en lazos de control tradicionales).

5.2.3 Resultados obtenidos

A continuación se mostrarán algunos de los resultados obtenidos en el laboratorio con el prototipo del microcontrolador y el prototipo de la grúa.

Control de posición y ángulo con desplazamiento

En la figura 5.4 podemos ver la evolución de la señal de control del motor junto con las señales que se leen de los dos sensores de la planta. Nótese que el ángulo casi no se perturba durante todo el recorrido del vehículo.

Control de posición y ángulo con oscilaciones de la carga

En este caso y partiendo de la posición de reposo de la planta, se ha introducido una perturbación externa sobre la carga. En la gráfica 5.5 se puede apreciar como el motor actúa en contraposición de las ocilaciones de la carga hasta que éstas se anulan. En la figura 5.6 se puede ver, a modo de comparación, como oscilaría la carga en el caso de no existir control alguno por parte del microcontrolador.

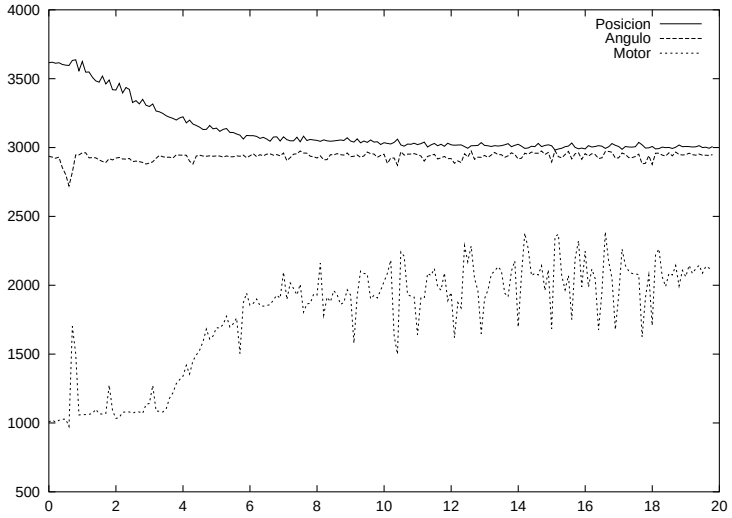


Figura 5.4: Control de posición y ángulo con desplazamiento

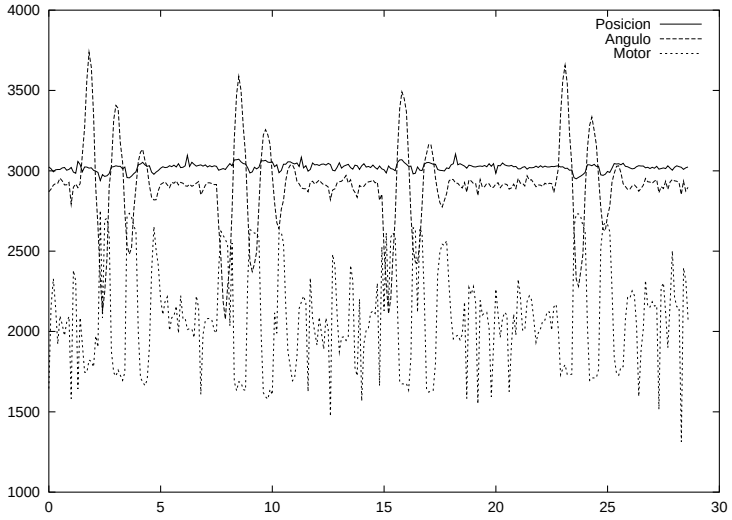


Figura 5.5: Control de posición y ángulo con oscilaciones de la carga

Control sólo de posición

En este caso hemos editado el código *fuzz* y hemos comentado las líneas correspondientes al control de la posición en función del ángulo de la carga y hemos ejecutado el nuevo código en el prototipo. Se puede ver en la gráfica 5.7 cómo se generan oscilaciones en la carga a partir del movimiento. Comparando la gráfica 5.4 con ésta, vemos que en la primera las oscilaciones de la carga son atenuadas rápidamente por el controlador.

Como se puede apreciar, la gráfica correspondiente a la señal del motor acusa constantes altibajos. Hay que señalar que estas alteraciones sólo se dieron en el momento de la medida y que se deben al ruido introducido por los sensores analógicos que se conectaron al PC con el fin de capturar los datos.

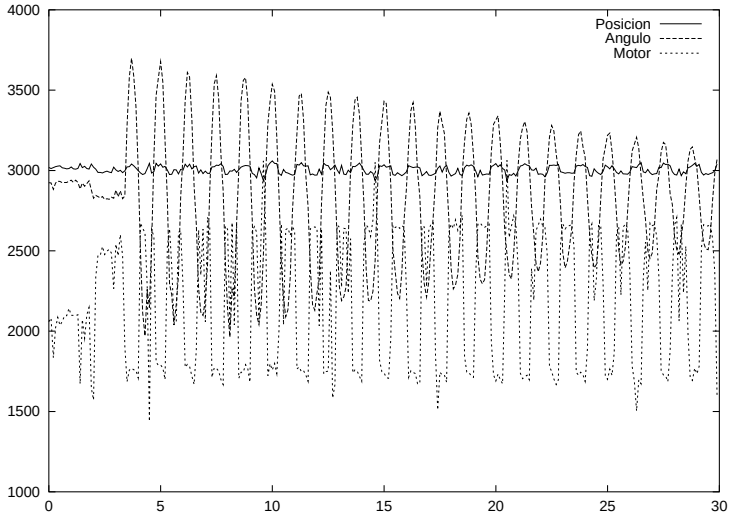


Figura 5.6: Oscilación libre de la carga

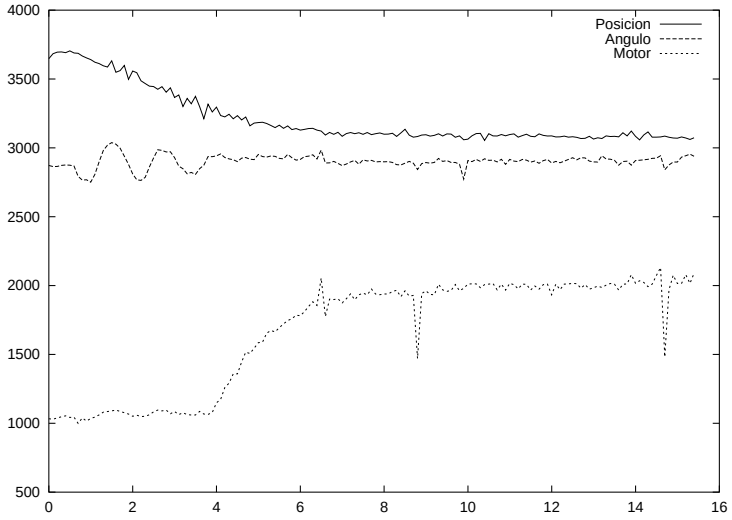


Figura 5.7: Control de posición con desplazamiento

La planta, cuando no se encuentra conectada a estos sensores situados en el PC, se comporta de forma totalmente suave al no interferir en las señales de los sensores del bucle de control.

En la figura 5.8 podemos ver una fotografía con todo el conjunto montado: el sistema de desarrollo junto con el vagón grúa.

5.3 Utilización de *fuzz* en lazos de control tradicionales

En el ejemplo de aplicación presentado, los datos de la planta se obtienen directamente de ésta. No hemos utilizado un lazo de control tradicional para implementar el control

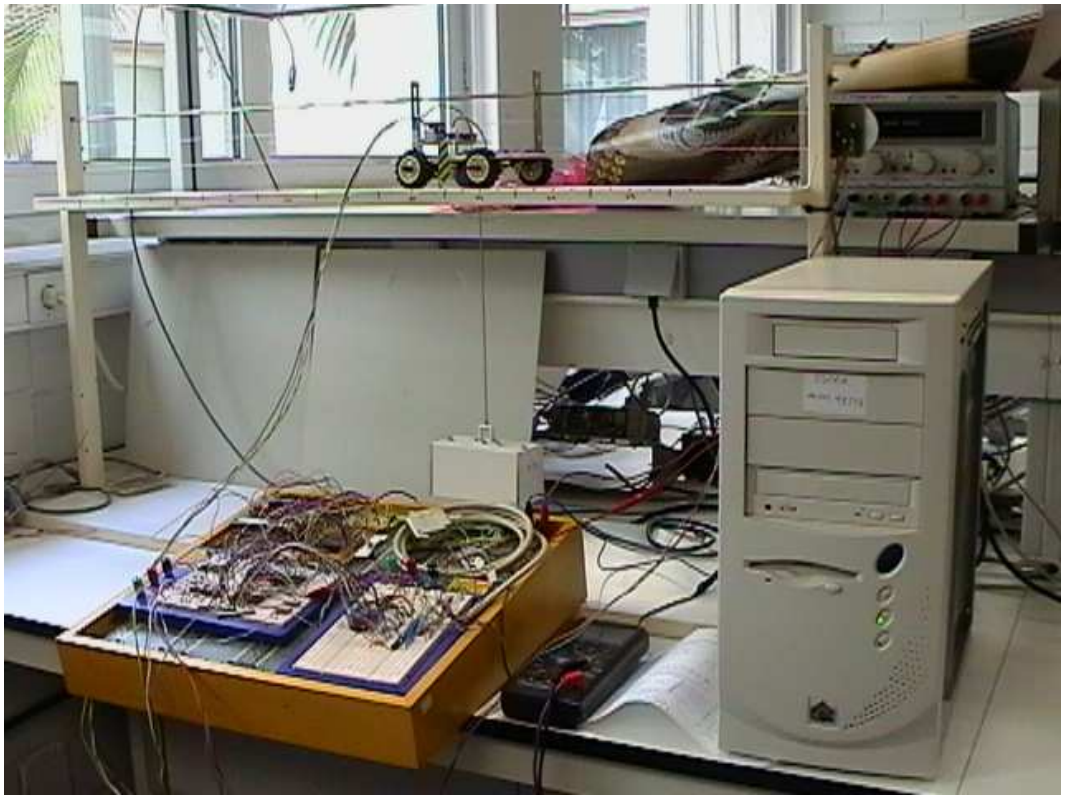


Figura 5.8: Fotografía de todo el conjunto

borroso ya que las entradas del prototipo no son los errores sino los valores directos que provienen de los sensores del vagón grúa. Ver la figura 5.9

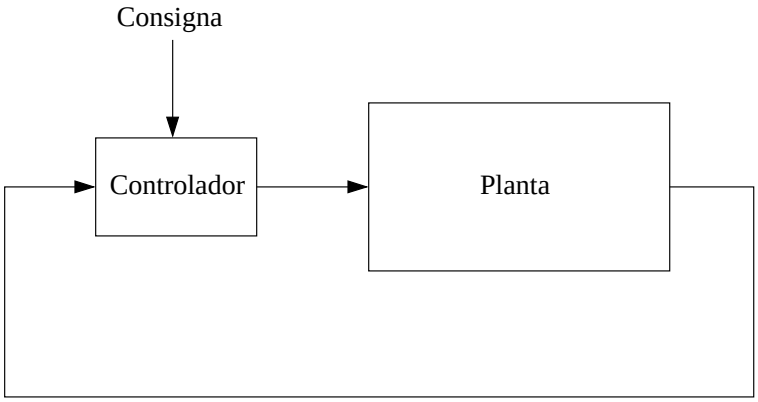


Figura 5.9: Lazo de control utilizado

En la figura 5.10 podemos ver lo que sería un lazo de control tradicional en el que las entradas al controlador son las señales de error (consigna - valores_salida_planta).

Para utilizar el preprocesador `microcyc_fuzz` utilizando lazos de control tradicional podemos optar por varias soluciones:

- Una solución hardware, basada en un circuito restador analógico.

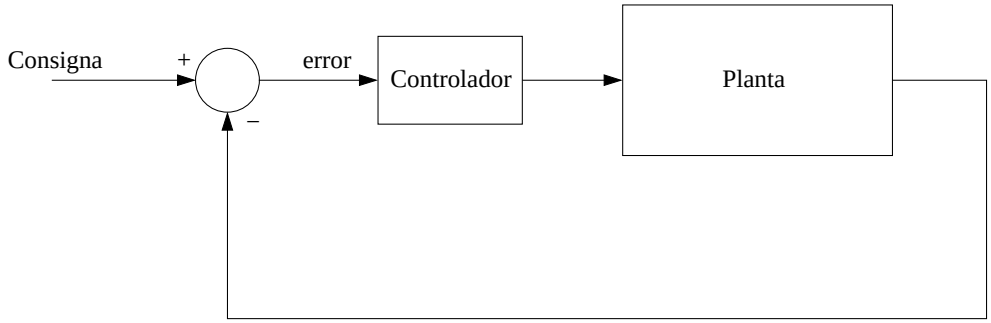


Figura 5.10: Lazo de control tradicional

- Una solución software, basada en el uso de variables borrosas con la cláusula `at anywhere`.

5.3.1 La solución hardware

Para implementar esta solución simplemente montaremos un circuito restador analógico en la entrada analógica por la que queramos medir el error. Ver la figura 5.11.

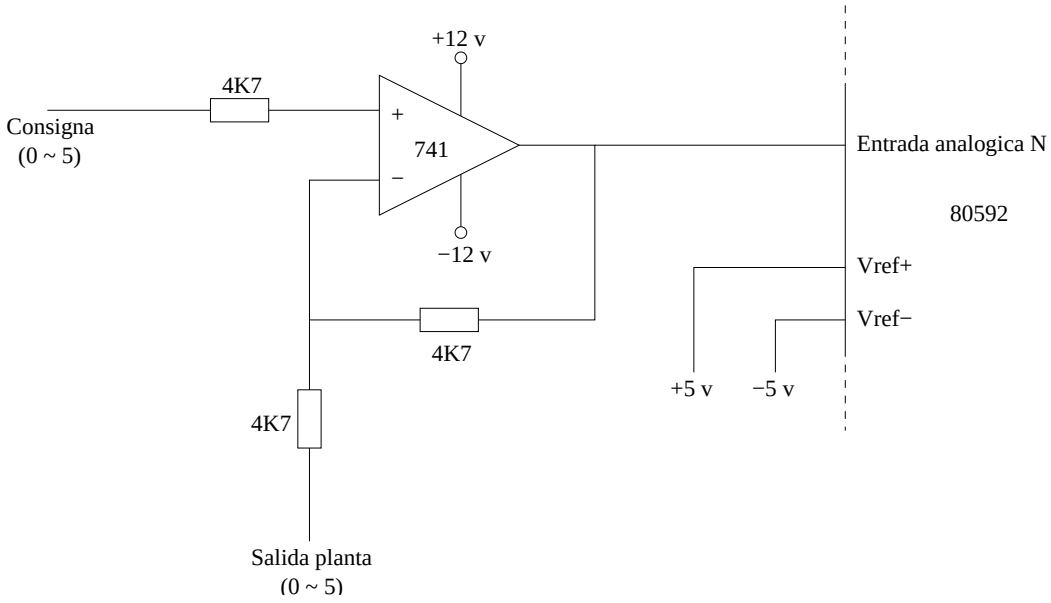


Figura 5.11: Restador analógico en la entrada del microcontrolador

Nótese como se han cambiado los voltajes de referencia del conversor AD del microcontrolador. Al estar el voltaje de consigna entre 0 y 5 voltios, y el valor de salida de la planta también entre 0 y 5 voltios; el resultado de la resta analógica dará unos resultados en el rango de -5 a +5 voltios. De esta forma, un valor de -5 voltios genera una configuración binaria 00000000 y un valor de +5 voltios una configuración binaria de 11111111.

En el código *fuzz* que generemos declararemos la variable de error de la forma usual:

. . .

```
var error at adc N = . . .
. . .
```

Siendo N la entrada analógica que hemos utilizado. De esta manera, a la hora de definir los conjuntos difusos, el error cero estaría en el punto 50.

5.3.2 La solución software

Mediante una solución software no tendremos que añadir circuitería externa al microcontrolador, aunque por contra, deberemos usar al menos 2 entradas analógicas por cada entrada de error (por una de las entradas se introducirá la consigna y por la otra entrada de cada par se introducirá el valor de salida de la planta). Ver la figura 5.12

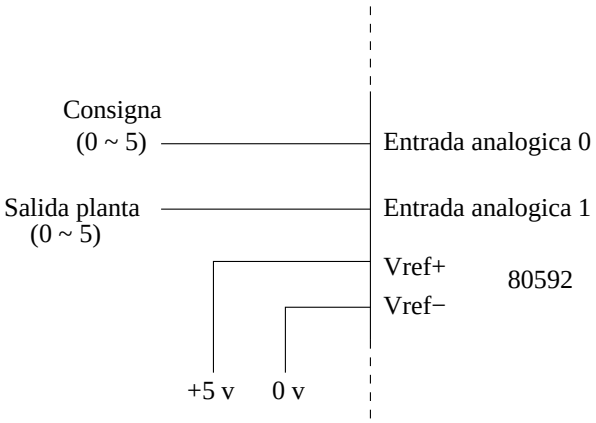


Figura 5.12: Ejemplo de conexionado para implementar una solución software

Para obtener la señal de error lo que debemos hacer es realizar la resta de las señales de entrada por software, una vez pasadas a valores numéricos por el conversor AD. El procedimiento es algo más trabajoso ya que requiere añadir código al fichero de salida del preprocesador `microcyc_fuzz`. Estos son los pasos a seguir:

1. Declaramos la variable difusa de error mediante la cláusula `at anywhere`, de esta forma el preprocesador sólo genera una variable de tipo `unsigned char` asociada a la variable difusa:

   ```
   error at anywhere = . . .
   ```

2. Definimos los conjuntos difusos y las reglas de forma usual y teniendo en cuenta que, al igual que en el caso de la solución hardware, un error 0 implica un valor de 50 a la hora de crear los conjuntos.

3. Una vez pasado el código *fuzz* por el preprocesador modificamos el código C resultante insertando el siguiente código dentro del bucle infinito de la función `main` y antes de la llamada a `fuzz_work`:

```
/* leemos las entradas analógicas */
fuzz_read_adc(0);
consigna = analog_input;
fuzz_read_adc(1);
salida_planta = analog_input;
/* restamos y escalamos */
error_value = (unsigned char) ((((int)consigna - (int)salida_planta)
                               + 5) / 2);
```

La función `fuzz_read_adc()` es una función que define previamente el preprocesador y no debe ser definida por el usuario. El parámetro que se le pasa es la entrada analógica que deseamos leer. La variable `analog_input` es también una variable definida por el preprocesador, y contiene el último valor convertido por el ADC. Las variables `consigna` y `salida_planta` son dos variables `unsigned char` que deberemos de definir a mano (a nivel local o global, como queramos).

La variable `error_value` es la variable `unsigned char` de C que crea el preprocesador a partir de la variable difusa `error` con la cláusula `at anywhere`. Como se puede apreciar en el código se trata simplemente de hacer la operación de resta y escalado que, de otra manera, tendríamos que hacer por hardware.

Inmediatamente después de estas líneas irá la llamada a la función `fuzz_work()`; esta función fuzzifica y trabaja con la variable `error_value` como si de una variable de entrada normal se tratase.

Apéndice A

El compilador SDCC

Extracto del manual del compilador SDCC. Se ha incluido únicamente información de interés para la programación del prototipo. El manual completo se encuentra en la documentación adjunta.

```
SDCC Compiler User Guide

Table of Contents

Introduction
    About SDCC
    Open Source
    Typographic conventions
    Compatibility with previous versions
    System Requirements
    Other Resources
    Wishes for the future
    Components of SDCC
        sdcc - The Compiler
        sdcpp (C-Preprocessor)
        asx8051, as-z80, as-gbz80, aslink, link-z80, link-gbz80 (The Assemblers and Link
        s51 - Simulator
        sdcdb - Source Level Debugger
Using SDCC
    Compiling
        Single Source File Projects
        Projects with Multiple Source Files
        Projects with Additional Libraries
    Command Line Options
        Processor Selection Options
        Preprocessor Options
```

Introduction

About SDCC

SDCC is a Freeware, retargettable, optimizing ANSI-C compiler by Sandeep Dutta designed for 8 bit Microprocessors. The current version targets Intel MCS51 based Microprocessors(8051,8052, etc), Zilog Z80 based MCUs, and the Dallas DS80C390 variant. It can be retargetted for other microprocessors, support for PIC, AVR and 186 is under development. The entire source code for the compiler is distributed under GPL. SDCC uses ASXXXX & ASLINK, a Freeware, retargettable assembler & linker. SDCC has extensive language extensions suitable for utilizing various microcontrollers and underlying hardware effectively.

In addition to the MCU specific optimizations SDCC also does
a host of standard optimizations like:

 global sub expression elimination,

 loop optimizations (loop invariant, strength reduction
  of induction variables and loop reversing),

 constant folding & propagation,

 copy propagation,

 dead code elimination

 jumptables for switch statements.

For the back-end SDCC uses a global register allocation scheme
which should be well suited for other 8 bit MCUs.

The peep hole optimizer uses a rule based substitution mechanism
which is MCU independent.

Supported data-types are:

 char (8 bits, 1 byte),

 short and int (16 bits, 2 bytes),

 long (32 bit, 4 bytes)

 float (4 byte IEEE).

The compiler also allows inline assembler code to be embedded
anywhere in a function. In addition, routines developed
in assembly can also be called.

SDCC also provides an option (--cyclomatic) to report the
relative complexity of a function. These functions can then
be further optimized, or hand coded in assembly if needed.

SDCC also comes with a companion source level debugger SDCDB,
the debugger currently uses ucSim a freeware simulator for
8051 and other micro-controllers.

The latest version can be downloaded from [http://sdcc.sourceforge.net/].

Open Source

All packages used in this compiler system are opensource
and freeware; source code for all the sub-packages (asxxxx
assembler/linker, pre-processor) is distributed with the
package. This documentation is maintained using a freeware
word processor (LyX).

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version
2, or (at your option) any later version. This program is
distributed in the hope that it will be useful, but WITHOUT
ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General
Public License for more details. You should have received
a copy of the GNU General Public License along with this
program; if not, write to the Free Software Foundation,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
In other words, you are welcome to use, share and improve
this program. You are forbidden to forbid anyone else to
use, share and improve what you give them. Help stamp out
software-hoarding!

Typographic conventions

Throughout this manual, we will use the following convention.
Commands you have to type in are printed in "sans serif".
Code samples are printed in typewriter font. Interesting
items and new terms are printed in italicised type.

Compatibility with previous versions

This version has numerous bug fixes compared with the previous
version. But we also introduced some incompatibilities with
older versions. Not just for the fun of it, but to make
the compiler more stable, efficient and ANSI compliant.

short is now equivalent to int (16 bits), it used to be
 equivalent to char (8 bits)

the default directory where include, library and documention
 files are stored is no in /usr/local/share

char type parameters to vararg functions are casted to
 int unless explicitly casted, e.g.:

```
    char a=3;
    printf ("%d %c\n", a, (char)a);
   will push a as an int and as a char resp.

 option --regextend has been removed

 option --noreparms has been removed

<pending: more incompatibilities?>

 System Requirements

What do you need before you start installation of SDCC? A
computer, and a desire to compute. The preferred method
of installation is to compile SDCC from source using GNU
gcc and make. For Windows some pre-compiled binary distributions
are available for your convenience. You should have some
experience with command line tools and compiler use.

 Other Resources

The SDCC home page at [http://sdcc.sourceforge.net/]
is a great place to find distribution sets. You can also
find links to the user mailing lists that offer help or
discuss SDCC with other SDCC users. Web links to other SDCC
related sites can also be found here. This document can
be found in the DOC directory of the source package as a
text or HTML file. Some of the other tools (simulator and
assembler) included with SDCC contain their own documentation
and can be found in the source distribution. If you want
the latest unreleased software, the complete source package
is available directly by anonymous CVS on cvs.sdcc.sourceforge.net.

 Wishes for the future

There are (and always will be) some things that could be
done. Here are some I can think of:


sdcc -c --model-large -o large _atoi.c (where large could
be a different basename or a directory)


char KernelFunction3(char p) at 0x340;

If you can think of some more, please send them to the list.
```

<pending: And then of course a proper index-table>

Components of SDCC

SDCC is not just a compiler, but a collection of tools by
various developers. These include linkers, assemblers, simulators
and other components. Here is a summary of some of the components.
Note that the included simulator and assembler have separate
documentation which you can find in the source package in
their respective directories. As SDCC grows to include support
for other processors, other packages from various developers
are included and may have their own sets of documentation.

You might want to look at the files which are installed in
<installdir>. At the time of this writing, we find the following
programs:

In <installdir>/bin:

sdcc - The compiler.

sdcpp - The C preprocessor.

asx8051 - The assembler for 8051 type processors.

as-z80, as-gbz80 - The Z80 and GameBoy Z80 assemblers.

aslink -The linker for 8051 type processors.

link-z80, link-gbz80 - The Z80 and GameBoy Z80 linkers.

s51 - The ucSim 8051 simulator.

sdcdb - The source debugger.

packihx - A tool to pack Intel hex files.

In <installdir>/share/sdcc/include

the include files

In <installdir>/share/sdcc/lib

the sources of the runtime library and the subdirs small
large and ds390 with the precompiled relocatables.

In <installdir>/share/sdcc/doc

the documentation

As development for other processors proceeds, this list will
expand to include executables to support processors like
AVR, PIC, etc.

sdcc - The Compiler

This is the actual compiler, it in turn uses the c-preprocessor
and invokes the assembler and linkage editor.

sdcpp (C-Preprocessor)

The preprocessor is a modified version of the GNU preprocessor.
The C preprocessor is used to pull in #include sources,
process #ifdef statements, #defines and so on.

asx8051, as-z80, as-gbz80, aslink, link-z80, link-gbz80
(The Assemblers and Linkage Editors)

This is retargettable assembler & linkage editor, it was
developed by Alan Baldwin. John Hartman created the version
for 8051, and I (Sandeep) have made some enhancements and
bug fixes for it to work properly with the SDCC.

s51 - Simulator

S51 is a freeware, opensource simulator developed by Daniel
Drotos ([mailto:drdani@mazsola.iit.uni-miskolc.hu]).
The simulator is built as part of the build process. For
more information visit Daniel's website at: [http://mazsola.iit.uni-miskolc.hu/~drdani/e

sdcdb - Source Level Debugger

Sdcdb is the companion source level debugger. The current
version of the debugger uses Daniel's Simulator S51, but
can be easily changed to use other simulators.

Using SDCC

Compiling

Single Source File Projects

For single source file 8051 projects the process is very
simple. Compile your programs with the following command

"sdcc sourcefile.c". This will compile, assemble and link
your source file. Output files are as follows

sourcefile.asm - Assembler source file created by the compiler
sourcefile.lst - Assembler listing file created by the Assembler
sourcefile.rst - Assembler listing file updated with linkedit
information, created by linkage editor
sourcefile.sym - symbol listing for the sourcefile, created
by the assembler
sourcefile.rel - Object file created by the assembler, input
to Linkage editor
sourcefile.map - The memory map for the load module, created
by the Linker
sourcefile.ihx - The load module in Intel hex format (you
can select the Motorola S19 format with --out-fmt-s19)
sourcefile.cdb - An optional file (with --debug) containing
debug information

Projects with Multiple Source Files

SDCC can compile only ONE file at a time. Let us for example
assume that you have a project containing the following
files:

foo1.c (contains some functions)
foo2.c (contains some more functions)
foomain.c (contains more functions and the function main)

The first two files will need to be compiled separately with
the commands:

```
sdcc -c foo1.c
sdcc -c foo2.c
```

Then compile the source file containing the main() function
and link the files together with the following command:

```
sdcc foomain.c foo1.rel foo2.rel
```

Alternatively, foomain.c can be separately compiled as well:

```
sdcc -c foomain.c
sdcc foomain.rel foo1.rel foo2.rel
```

The file containing the main() function must be the first
file specified in the command line, since the linkage editor
processes file in the order they are presented to it.

Projects with Additional Libraries

Some reusable routines may be compiled into a library, see
the documentation for the assembler and linkage editor (which
are in <installdir>/share/sdcc/doc) for how to create a
.lib library file. Libraries created in this manner can
be included in the command line. Make sure you include the
-L <library-path> option to tell the linker where to look
for these files if they are not in the current directory.
Here is an example, assuming you have the source file foomain.c
and a library foolib.lib in the directory mylib (if that
is not the same as your current project):

```
sdcc foomain.c foolib.lib -L mylib
```

Note here that mylib must be an absolute path name.

The most efficient way to use libraries is to keep seperate
modules in seperate source files. The lib file now should
name all the modules.rel files. For an example see the standard
library file libsdcc.lib in the directory <installdir>/share/lib/small.

Command Line Options

Processor Selection Options

-mmcs51 Generate code for the MCS51 (8051) family of processors.
This is the default processor target.

-mds390 Generate code for the DS80C390 processor.

-mz80 Generate code for the Z80 family of processors.

-mgbz80 Generate code for the GameBoy Z80 processor.

-mavr Generate code for the Atmel AVR processor(In development,
not complete).

-mpic14 Generate code for the PIC 14-bit processors(In development,
not complete).

-mtlcs900h Generate code for the Toshiba TLCS-900H processor(In
development, not complete).

Preprocessor Options

-I<path> The additional location where the pre processor
will look for <..h> or "..h"
files.

-D<macro[=value]> Command line definition of macros. Passed
to the pre processor.

-M Tell the preprocessor to output a rule suitable for make
describing the dependencies of each object file. For each
source file, the preprocessor outputs one make-rule whose
target is the object file name for that source file and
whose dependencies are all the files ‘#include’d in it.
This rule may be a single line or may be continued with
‘\’-newline if it is long. The list of rules is printed on
standard output instead of the preprocessed C program. ‘-M’
implies ‘-E’.

-C Tell the preprocessor not to discard comments. Used with
the ‘-E’ option.

-MM Like ‘-M’ but the output mentions only the user header
files included with ‘#include "file"’.
System header files included with ‘#include <file>’ are
omitted.

-Aquestion(answer) Assert the answer answer for question,
in case it is tested with a preprocessor conditional such
as ‘#if #question(answer)’. ‘-A-’ disables the standard
assertions that normally describe the target machine.

-Aquestion (answer) Assert the answer answer for question,
in case it is tested with a preprocessor conditional such
as ‘#if #question(answer)’. ‘-A-’ disables the standard
assertions that normally describe the target machine.

-Umacro Undefine macro macro. ‘-U’ options are evaluated
after all ‘-D’ options, but before any ‘-include’ and ‘-imacros’
options.

-dM Tell the preprocessor to output only a list of the macro
definitions that are in effect at the end of preprocessing.
Used with the ‘-E’ option.

-dD Tell the preprocessor to pass all macro definitions into

the output, in their proper sequence in the rest of the output.

-dN Like ‘-dD’ except that the macro arguments and contents are omitted. Only ‘#define name’ is included in the output.

Linker Options

-L --lib-path <absolute path to additional libraries> This option is passed to the linkage editor’s additional libraries search path. The path name must be absolute. Additional library files may be specified in the command line. See section Compiling programs for more details.

--xram-loc<Value> The start location of the external ram, default value is 0. The value entered can be in Hexadecimal or Decimal format, e.g.: --xram-loc 0x8000 or --xram-loc 32768.

--code-loc<Value> The start location of the code segment, default value 0. Note when this option is used the interrupt vector table is also relocated to the given address. The value entered can be in Hexadecimal or Decimal format, e.g.: --code-loc 0x8000 or --code-loc 32768.

--stack-loc<Value> The initial value of the stack pointer. The default value of the stack pointer is 0x07 if only register bank 0 is used, if other register banks are used then the stack pointer is initialized to the location above the highest register bank used. eg. if register banks 1 & 2 are used the stack pointer will default to location 0x18. The value entered can be in Hexadecimal or Decimal format, eg. --stack-loc 0x20 or --stack-loc 32. If all four register banks are used the stack will be placed after the data segment (equivalent to --stack-after-data)

--stack-after-data This option will cause the stack to be located in the internal ram after the data segment.

--data-loc<Value> The start location of the internal ram data segment, the default value is 0x30.The value entered can be in Hexadecimal or Decimal format, eg. --data-loc 0x20 or --data-loc 32.

--idata-loc<Value> The start location of the indirectly addressable internal ram, default value is 0x80. The value entered can be in Hexadecimal or Decimal format, eg. --idata-loc 0x88

or --idata-loc 136.

--out-fmt-ihx The linker output (final object code) is in
Intel Hex format. (This is the default option).

--out-fmt-s19 The linker output (final object code) is in
Motorola S19 format.

MCS51 Options

--model-large Generate code for Large model programs see
section Memory Models for more details. If this option is
used all source files in the project should be compiled
with this option. In addition the standard library routines
are compiled with small model, they will need to be recompiled.

--model-small Generate code for Small Model programs see
section Memory Models for more details. This is the default
model.

DS390 Options

--model-flat24 Generate 24-bit flat mode code. This is the
one and only that the ds390 code generator supports right
now and is default when using -mds390. See section Memory
Models for more details.

--stack-10bit Generate code for the 10 bit stack mode of
the Dallas DS80C390 part. This is the one and only that
the ds390 code generator supports right now and is default
when using -mds390. In this mode, the stack is located in
the lower 1K of the internal RAM, which is mapped to 0x400000.
Note that the support is incomplete, since it still uses
a single byte as the stack pointer. This means that only
the lower 256 bytes of the potential 1K stack space will
actually be used. However, this does allow you to reclaim
the precious 256 bytes of low RAM for use for the DATA and
IDATA segments. The compiler will not generate any code
to put the processor into 10 bit stack mode. It is important
to ensure that the processor is in this mode before calling
any re-entrant functions compiled with this option. In principle,
this should work with the --stack-auto option, but that
has not been tested. It is incompatible with the --xstack
option. It also only makes sense if the processor is in
24 bit contiguous addressing mode (see the --model-flat24
option).

Optimization Options

--nogcse Will not do global subexpression elimination, this option may be used when the compiler creates undesirably large stack/data spaces to store compiler temporaries. A warning message will be generated when this happens and the compiler will indicate the number of extra bytes it allocated. It recommended that this option NOT be used, #pragma NOGCSE can be used to turn off global subexpression elimination for a given function only.

--noinvariant Will not do loop invariant optimizations, this may be turned off for reasons explained for the previous option. For more details of loop optimizations performed see section Loop Invariants.It recommended that this option NOT be used, #pragma NOINVARIANT can be used to turn off invariant optimizations for a given function only.

--noinduction Will not do loop induction optimizations, see section strength reduction for more details.It is recommended that this option is NOT used, #pragma NOINDUCTION can be used to turn off induction optimizations for a given function only.

--nojtbound Will not generate boundary condition check when switch statements are implemented using jump-tables. See section Switch Statements for more details. It is recommended that this option is NOT used, #pragma NOJTBOUND can be used to turn off boundary checking for jump tables for a given function only.

--noloopreverse Will not do loop reversal optimization.

Other Options

-c --compile-only will compile and assemble the source, but will not call the linkage editor.

-E Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

--stack-auto All functions in the source file will be compiled as reentrant, i.e. the parameters and local variables will be allocated on the stack. see section Parameters and Local Variables for more details. If this option is used all source files in the project should be compiled with this option.

--xstack Uses a pseudo stack in the first 256 bytes in the
external ram for allocating variables and passing parameters.
See section on external stack for more details.

--callee-saves function1[,function2][,function3].... The
compiler by default uses a caller saves convention for register
saving across function calls, however this can cause unneccessary
register pushing & popping when calling small functions
from larger functions. This option can be used to switch
the register saving convention for the function names specified.
The compiler will not save registers when calling these
functions, no extra code will be generated at the entry
& exit for these functions to save & restore the registers
used by these functions, this can SUBSTANTIALLY reduce code
& improve run time performance of the generated code. In
the future the compiler (with interprocedural analysis)
will be able to determine the appropriate scheme to use
for each function call. DO NOT use this option for built-in
functions such as _muluint..., if this option is used for
a library function the appropriate library function needs
to be recompiled with the same option. If the project consists
of multiple source files then all the source file should
be compiled with the same --callee-saves option string.
Also see #pragma CALLEE-SAVES.

--debug When this option is used the compiler will generate
debug information, that can be used with the SDCDB. The
debug information is collected in a file with .cdb extension.
For more information see documentation for SDCDB.

--regextend This option is obsolete and isn't supported
anymore.

--noregparms This option is obsolete and isn't supported
anymore.

--peep-file<filename> This option can be used to use additional
rules to be used by the peep hole optimizer. See section
Peep Hole optimizations for details on how to write these
rules.

-S Stop after the stage of compilation proper; do not assemble.
The output is an assembler code file for the input file
specified.

-Wa_asmOption[,asmOption]... Pass the asmOption to the assembler.

-Wl_linkOption[,linkOption]... Pass the linkOption to the
linker.

--int-long-reent Integer (16 bit) and long (32 bit) libraries
have been compiled as reentrant. Note by default these libraries
are compiled as non-reentrant. See section Installation
for more details.

--cyclomatic This option will cause the compiler to generate
an information message for each function in the source file.
The message contains some important information about the
function. The number of edges and nodes the compiler detected
in the control flow graph of the function, and most importantly
the cyclomatic complexity see section on Cyclomatic Complexity
for more details.

--float-reent Floating point library is compiled as reentrant.See
section Installation for more details.

--nooverlay The compiler will not overlay parameters and
local variables of any function, see section Parameters
and local variables for more details.

--main-return This option can be used when the code generated
is called by a monitor program. The compiler will generate
a 'ret' upon return from the 'main' function. The default
option is to lock up i.e. generate a 'ljmp '.

--no-peep Disable peep-hole optimization.

--peep-asm Pass the inline assembler code through the peep
hole optimizer. This can cause unexpected changes to inline
assembler code, please go through the peephole optimizer
rules defined in the source file tree '<target>/peeph.def'
before using this option.

--iram-size<Value> Causes the linker to check if the interal
ram usage is within limits of the given value.

--nostdincl This will prevent the compiler from passing on
the default include path to the preprocessor.

--nostdlib This will prevent the compiler from passing on
the default library path to the linker.

--verbose Shows the various actions the compiler is performing.

-V Shows the actual commands the compiler is executing.

Intermediate Dump Options

The following options are provided for the purpose of retargetting
and debugging the compiler. These provided a means to dump
the intermediate code (iCode) generated by the compiler
in human readable form at various stages of the compilation
process.

--dumpraw This option will cause the compiler to dump the
intermediate code into a file of named <source filename>.dumpraw
just after the intermediate code has been generated for
a function, i.e. before any optimizations are done. The
basic blocks at this stage ordered in the depth first number,
so they may not be in sequence of execution.

--dumpgcse Will create a dump of iCode's, after global subexpression
elimination, into a file named <source filename>.dumpgcse.

--dumpdeadcode Will create a dump of iCode's, after deadcode
elimination, into a file named <source filename>.dumpdeadcode.

--dumploop Will create a dump of iCode's, after loop optimizations,
into a file named <source filename>.dumploop.

--dumprange Will create a dump of iCode's, after live range
analysis, into a file named <source filename>.dumprange.

--dumlrange Will dump the life ranges for all symbols.

--dumpregassign Will create a dump of iCode's, after register
assignment, into a file named <source filename>.dumprassgn.

--dumplrange Will create a dump of the live ranges of iTemp's

--dumpall Will cause all the above mentioned dumps to be
created.

MCS51/DS390 Storage Class Language Extensions

In addition to the ANSI storage classes SDCC allows the following
MCS51 specific storage classes.

xdata

```
Variables declared with this storage class will be placed
in the extern RAM. This is the default storage class for
Large Memory model, e.g.:

xdata unsigned char xduc;

 data

This is the default storage class for Small Memory model.
Variables declared with this storage class will be allocated
in the internal RAM, e.g.:

data int iramdata;

 idata

Variables declared with this storage class will be allocated
into the indirectly addressable portion of the internal
ram of a 8051, e.g.:

idata int idi;

 bit

This is a data-type and a storage class specifier. When a
variable is declared as a bit, it is allocated into the
bit addressable memory of 8051, e.g.:

bit iFlag;

 sfr / sbit

Like the bit keyword, sfr / sbit signifies both a data-type
and storage class, they are used to describe the special
function registers and special bit variables of a 8051,
eg:

sfr at 0x80 P0; /* special function register P0 at location
0x80 */
sbit at 0xd7 CY; /* CY (Carry Flag) */

 Pointers

SDCC allows (via language extensions) pointers to explicitly
point to any of the memory spaces of the 8051. In addition
to the explicit pointers, the compiler also allows a _generic
class of pointers which can be used to point to any of the
```

```
memory spaces.

Pointer declaration examples:

/* pointer physically in xternal ram pointing to object in
internal ram */
data unsigned char * xdata p;

/* pointer physically in code rom pointing to data in xdata
space */
xdata unsigned char * code p;

/* pointer physically in code space pointing to data in code
space */
code unsigned char * code p;

/* the folowing is a generic pointer physically located in
xdata space */
char * xdata p;

Well you get the idea.

For compatibility with the previous version of the compiler,
the following syntax for pointer declaration is still supported
but will disappear int the near future.

unsigned char _xdata *ucxdp; /* pointer to data in external
ram */
unsigned char _data  *ucdp ; /* pointer
to data in internal ram */
unsigned char _code  *uccp ; /* pointer
to data in R/O code space */
unsigned char _idata *uccp;  /*
pointer to upper 128 bytes of ram */

All unqualified pointers are treated as 3-byte (4-byte for
the ds390) generic pointers. These type of pointers can
also to be explicitly declared.

unsigned char _generic *ucgp;

The highest order byte of the generic pointers contains the
data space information. Assembler support routines are called
whenever data is stored or retrieved using generic pointers.
These are useful for developing reusable library routines.
Explicitly specifying the pointer type will generate the
most efficient code. Pointers declared using a mixture of
```

OLD and NEW style could have unpredictable results.

Parameters & Local Variables

Automatic (local) variables and parameters to functions can either be placed on the stack or in data-space. The default action of the compiler is to place these variables in the internal RAM (for small model) or external RAM (for Large model). This in fact makes them static so by default functions are non-reentrant.

They can be placed on the stack either by using the --stack-auto compiler option or by using the reentrant keyword in the function declaration, e.g.:

```
unsigned char foo(char i) reentrant
{
...
}
```

Since stack space on 8051 is limited, the reentrant keyword or the --stack-auto option should be used sparingly. Note that the reentrant keyword just means that the parameters & local variables will be allocated to the stack, it does not mean that the function is register bank independent.

Local variables can be assigned storage classes and absolute addresses, e.g.:

```
unsigned char foo() {
    xdata unsigned char i;
    bit bvar;
    data at 0x31 unsiged char j;
    ...
}
```

In the above example the variable i will be allocated in the external ram, bvar in bit addressable space and j in internal ram. When compiled with --stack-auto or when a function is declared as reentrant this can only be done for static variables.

Parameters however are not allowed any storage class, (storage classes for parameters will be ignored), their allocation is governed by the memory model in use, and the reentrancy options.

For non-reentrant functions SDCC will try to reduce internal ram space usage by overlaying parameters and local variables of a function (if possible). Parameters and local variables of a function will be allocated to an overlayable segment if the function has no other function calls and the function is non-reentrant and the memory model is small. If an explicit storage class is specified for a local variable, it will NOT be overlayed.

Note that the compiler (not the linkage editor) makes the decision for overlaying the data items. Functions that are called from an interrupt service routine should be preceded by a #pragma NOOVERLAY if they are not reentrant.

Also note that the compiler does not do any processing of inline assembler code, so the compiler might incorrectly assign local variables and parameters of a function into the overlay segment if the inline assembler code calls other c-functions that might use the overlay. In that case the #pragma NOOVERLAY should be used.

Parameters and Local variables of functions that contain 16 or 32 bit multiplication or division will NOT be overlayed since these are implemented using external functions, e.g.:

```
#pragma SAVE
#pragma NOOVERLAY
void set_error(unsigned char errcd)
{
    P3 = errcd;
}
#pragma RESTORE

void some_isr () interrupt 2 using 1
{
    ...
    set_error(10);
    ...
}
```

In the above example the parameter errcd for the function set_error would be assigned to the overlayable segment if the #pragma NOOVERLAY was not present, this could cause unpredictable runtime behavior when called from an ISR. The #pragma NOOVERLAY ensures that

the parameters and local variables for the function are
NOT overlayed.

Interrupt Service Routines

SDCC allows interrupt service routines to be coded in C,
with some extended keywords.

```
void timer_isr (void) interrupt 2 using 1
{
..
}
```

The number following the interrupt keyword is the interrupt
number this routine will service. The compiler will insert
a call to this routine in the interrupt vector table for
the interrupt number specified. The using keyword is used
to tell the compiler to use the specified register bank
(8051 specific) when generating code for this function.
Note that when some function is called from an interrupt
service routine it should be preceded by a #pragma NOOVERLAY
if it is not reentrant. A special note here, int (16 bit)
and long (32 bit) integer division, multiplication & modulus
operations are implemented using external support routines
developed in ANSI-C, if an interrupt service routine needs
to do any of these operations then the support routines
(as mentioned in a following section) will have to be recompiled
using the --stack-auto option and the source file will need
to be compiled using the --int-long-rent compiler option.

If you have multiple source files in your project, interrupt
service routines can be present in any of them, but a prototype
of the isr MUST be present or included in the file that
contains the function main.

Interrupt Numbers and the corresponding address & descriptions
for the Standard 8051 are listed below. SDCC will automatically
adjust the interrupt vector table to the maximum interrupt
number specified.

Interrupt #	Description	Vector Address
0	External 0	0x0003

	1	Timer 0	0x000B
	2	External 1	0x0013
	3	Timer 1	0x001B
	4	Serial	0x0023

If the interrupt service routine is defined without using a register bank or with register bank 0 (using 0), the compiler will save the registers used by itself on the stack upon entry and restore them at exit, however if such an interrupt service routine calls another function then the entire register bank will be saved on the stack. This scheme may be advantageous for small interrupt service routines which have low register usage.

If the interrupt service routine is defined to be using a specific register bank then only a, b & dptr are save and restored, if such an interrupt service routine calls another function (using another register bank) then the entire register bank of the called function will be saved on the stack. This scheme is recommended for larger interrupt service routines.

Calling other functions from an interrupt service routine is not recommended, avoid it if possible.

Also see the _naked modifier.

Critical Functions

A special keyword may be associated with a function declaring it as critical. SDCC will generate code to disable all interrupts upon entry to a critical function and enable them back before returning. Note that nesting critical functions may cause unpredictable results.

```
int foo () critical
{
...
...
}
```

The critical attribute maybe used with other attributes like

reentrant.

Naked Functions

A special keyword may be associated with a function declaring
it as _naked. The _naked function modifier attribute prevents
the compiler from generating prologue and epilogue code
for that function. This means that the user is entirely
responsible for such things as saving any registers that
may need to be preserved, selecting the proper register
bank, generating the return instruction at the end, etc.
Practically, this means that the contents of the function
must be written in inline assembler. This is particularly
useful for interrupt functions, which can have a large (and
often unnecessary) prologue/epilogue. For example, compare
the code generated by these two functions:

```
data unsigned char counter;
void simpleInterrupt(void) interrupt 1
{
    counter++;
}

void nakedInterrupt(void) interrupt 2 _naked
{
    _asm
      inc     _counter
      reti    ;
MUST explicitly include ret in _naked function.
    _endasm;
}
```

For an 8051 target, the generated simpleInterrupt looks like:

```
_simpleIterrupt:
    push    acc
    push    b
    push    dpl
    push    dph
    push    psw
    mov     psw,#0x00
    inc     _counter
    pop     psw
    pop     dph
    pop     dpl
    pop     b
    pop     acc
```

```
    reti
```

whereas nakedInterrupt looks like:

```
_nakedInterrupt:
    inc    _counter
    reti   ; MUST explicitly
include ret(i) in _naked function.
```

While there is nothing preventing you from writing C code
inside a _naked function, there are many ways to shoot yourself
in the foot doing this, and is is recommended that you stick
to inline assembler.

Functions using private banks

The using attribute (which tells the compiler to use a register
bank other than the default bank zero) should only be applied
to interrupt functions (see note 1 below). This will in
most circumstances make the generated ISR code more efficient
since it will not have to save registers on the stack.

The using attribute will have no effect on the generated
code for a non-interrupt function (but may occasionally
be useful anyway([footnote] possible exception: if a function is called ONLY
from 'interrupt' functions using a particular bank, it can
be declared with the same 'using' attribute as the calling
'interrupt' functions. For instance, if you have several
ISRs using bank one, and all of them call memcpy(), it might
make sense to create a specialized version of memcpy() 'using
1', since this would prevent the ISR from having to save
bank zero to the stack on entry and switch to bank zero
before calling the function)).
(pending: I don't think this has been done yet)

An interrupt function using a non-zero bank will assume that
it can trash that register bank, and will not save it. Since
high-priority interrupts can interrupt low-priority ones
on the 8051 and friends, this means that if a high-priority
ISR using a particular bank occurs while processing a low-priority
ISR using the same bank, terrible and bad things can happen.
To prevent this, no single register bank should be used
by both a high priority and a low priority ISR. This is
probably most easily done by having all high priority ISRs
use one bank and all low priority ISRs use another. If you
have an ISR which can change priority at runtime, you're
on your own: I suggest using the default bank zero and taking

the small performance hit.

It is most efficient if your ISR calls no other functions.
If your ISR must call other functions, it is most efficient
if those functions use the same bank as the ISR (see note
1 below); the next best is if the called functions use bank
zero. It is very inefficient to call a function using a
different, non-zero bank from an ISR.

Absolute Addressing

Data items can be assigned an absolute address with the at
<address> keyword, in addition to a storage class, e.g.:

```
xdata at 0x8000 unsigned char PORTA_8255 ;
```

In the above example the PORTA_8255 will be allocated to
the location 0x8000 of the external ram. Note that this
feature is provided to give the programmer access to memory
mapped devices attached to the controller. The compiler
does not actually reserve any space for variables declared
in this way (they are implemented with an equate in the
assembler). Thus it is left to the programmer to make sure
there are no overlaps with other variables that are declared
without the absolute address. The assembler listing file
(.lst) and the linker output files (.rst) and (.map) are
a good places to look for such overlaps.

Absolute address can be specified for variables in all storage
classes, e.g.:

```
bit at 0x02 bvar;
```

The above example will allocate the variable at offset 0x02
in the bit-addressable space. There is no real advantage
to assigning absolute addresses to variables in this manner,
unless you want strict control over all the variables allocated.

Startup Code

The compiler inserts a call to the C routine _sdcc__external__startup()
at the start of the CODE area. This routine is in the runtime
library. By default this routine returns 0, if this routine
returns a non-zero value, the static & global variable initialization
will be skipped and the function main will be invoked Other
wise static & global variables will be initialized before
the function main is invoked. You could add a _sdcc__external__startup()

routine to your program to override the default if you need
to setup hardware or perform some other critical operation
prior to static & global variable initialization.

Inline Assembler Code

SDCC allows the use of in-line assembler with a few restriction
as regards labels. All labels defined within inline assembler
code has to be of the form nnnnn$ where nnnn is a number
less than 100 (which implies a limit of utmost 100 inline
assembler labels per function). It is strongly recommended
that each assembly instruction (including labels) be placed
in a separate line (as the example shows). When the --peep-asm
command line option is used, the inline assembler code will
be passed through the peephole optimizer. This might cause
some unexpected changes in the inline assembler code. Please
go throught the peephole optimizer rules defined in file
SDCCpeeph.def carefully before using this option.

```
_asm
    mov     b,#10

00001$:
    djnz    b,00001$

_endasm ;
```

The inline assembler code can contain any valid code understood
by the assembler, this includes any assembler directives
and comment lines. The compiler does not do any validation
of the code within the _asm ... _endasm; keyword pair.

Inline assembler code cannot reference any C-Labels, however
it can reference labels defined by the inline assembler,
e.g.:

```
foo() {
    /* some c code */
    _asm
      ; some assembler code
      ljmp $0003
    _endasm;
    /* some more c code */
clabel:  /* inline assembler cannot reference
this label */
    _asm
    $0003: ;label (can be reference by inline assembler
```

```
only)
    _endasm ;
    /* some more c code */
}
```

In other words inline assembly code can access labels defined in inline assembly within the scope of the funtion.

The same goes the other way, ie. labels defines in inline assembly CANNOT be accessed by C statements.

int(16 bit) and long (32 bit) Support

For signed & unsigned int (16 bit) and long (32 bit) variables, division, multiplication and modulus operations are implemented by support routines. These support routines are all developed in ANSI-C to facilitate porting to other MCUs, although some model specific assembler optimations are used. The following files contain the described routine, all of them can be found in <installdir>/share/sdcc/lib.

<pending: tabularise this>

_mulsint.c - signed 16 bit multiplication (calls _muluint)
_muluint.c - unsigned 16 bit multiplication
_divsint.c - signed 16 bit division (calls _divuint)
_divuint.c - unsigned 16 bit division
_modsint.c - signed 16 bit modulus (call _moduint)
_moduint.c - unsigned 16 bit modulus
_mulslong.c - signed 32 bit multiplication (calls _mululong)
_mululong.c - unsigned32 bit multiplication
_divslong.c - signed 32 division (calls _divulong)
_divulong.c - unsigned 32 division
_modslong.c - signed 32 bit modulus (calls _modulong)
_modulong.c - unsigned 32 bit modulus

Since they are compiled as non-reentrant, interrupt service routines should not do any of the above operations. If this is unavoidable then the above routines will need to be compiled with the --stack-auto option, after which the source program will have to be compiled with --int-long-rent option.

Floating Point Support

SDCC supports IEEE (single precision 4bytes) floating point numbers.The floating point support routines are derived from gcc's floatlib.c and consists of the following routines:

<pending: tabularise this>

_fsadd.c - add floating point numbers
_fssub.c - subtract floating point numbers
_fsdiv.c - divide floating point numbers
_fsmul.c - multiply floating point numbers
_fs2uchar.c - convert floating point to unsigned char
_fs2char.c - convert floating point to signed char
_fs2uint.c - convert floating point to unsigned int
_fs2int.c - convert floating point to signed int
_fs2ulong.c - convert floating point to unsigned long
_fs2long.c - convert floating point to signed long
_uchar2fs.c - convert unsigned char to floating point
_char2fs.c - convert char to floating point number
_uint2fs.c - convert unsigned int to floating point
_int2fs.c - convert int to floating point numbers
_ulong2fs.c - convert unsigned long to floating point number
_long2fs.c - convert long to floating point number

Note if all these routines are used simultaneously the data
space might overflow. For serious floating point usage it
is strongly recommended that the large model be used.

MCS51 Memory Models

SDCC allows two memory models for MCS51 code, small and large.
Modules compiled with different memory models should never
be combined together or the results would be unpredictable.
The library routines supplied with the compiler are compiled
as both small and large. The compiled library modules are
contained in seperate directories as small and large so
that you can link to either set.

When the large model is used all variables declared without
a storage class will be allocated into the external ram,
this includes all parameters and local variables (for non-reentrant
functions). When the small model is used variables without
storage class are allocated in the internal ram.

Judicious usage of the processor specific storage classes
and the 'reentrant' function type will yield much more efficient
code, than using the large model. Several optimizations
are disabled when the program is compiled using the large
model, it is therefore strongly recommdended that the small
model be used unless absolutely required.

DS390 Memory Models

The only model supported is Flat 24. This generates code for the 24 bit contiguous addressing mode of the Dallas DS80C390 part. In this mode, up to four meg of external RAM or code space can be directly addressed. See the data sheets at www.dalsemi.com for further information on this part.

In older versions of the compiler, this option was used with the MCS51 code generator (-mmcs51). Now, however, the '390 has it's own code generator, selected by the -mds390 switch.

Note that the compiler does not generate any code to place the processor into 24 bitmode (although tinibios in the ds390 libraries will do that for you). If you don't use tinibios, the boot loader or similar code must ensure that the processor is in 24 bit contiguous addressing mode before calling the SDCC startup code.

Like the --model-large option, variables will by default be placed into the XDATA segment.

Segments may be placed anywhere in the 4 meg address space using the usual --*-loc options. Note that if any segments are located above 64K, the -r flag must be passed to the linker to generate the proper segment relocations, and the Intel HEX output format must be used. The -r flag can be passed to the linker by using the option -Wl-r on the sdcc command line. However, currently the linker can not handle code segments > 64k.

Defines Created by the Compiler

The compiler creates the following #defines.

SDCC - this Symbol is always defined.

SDCC_mcs51 or SDCC_ds390 or SDCC_z80, etc - depending on the model used (e.g.: -mds390)

__mcs51 or __ds390 or __z80, etc - depending on the model used (e.g. -mz80)

SDCC_STACK_AUTO - this symbol is defined when --stack-auto option is used.

SDCC_MODEL_SMALL - when --model-small is used.

SDCC_MODEL_LARGE - when --model-large is used.

SDCC_USE_XSTACK - when --xstack option is used.

SDCC_STACK_TENBIT - when -mds390 is used

SDCC_MODEL_FLAT24 - when -mds390 is used

Acknowledgments

Sandeep Dutta (sandeep.dutta@usa.net) - SDCC, the compiler,
MCS51 code generator, Debugger, AVR port
Alan Baldwin (baldwin@shop-pdp.kent.edu) - Initial version
of ASXXXX & ASLINK.
John Hartman (jhartman@compuserve.com) - Porting ASXXX &
ASLINK for 8051
Dmitry S. Obukhov (dso@usa.net) - malloc & serial i/o routines.

Daniel Drotos (drdani@mazsola.iit.uni-miskolc.hu) - for his
Freeware simulator
Malini Dutta(malini_dutta@hotmail.com) - my wife for her
patience and support.
Unknown - for the GNU C - preprocessor.
Michael Hope - The Z80 and Z80GB port, 186 development
Kevin Vigor - The DS390 port.
Johan Knol - Lots of fixes and enhancements, DS390/TINI libs.
Scott Datallo - The PIC port.

Thanks to all the other volunteer developers who have helped
with coding, testing, web-page creation, distribution sets,
etc. You know who you are :-)

This document was initially written by Sandeep Dutta

Apéndice B

El software del emulador: ihxwrite e ihxtest

Listado completo de los programas `ihxwrite` e `ihxtest`, para la programación y el testeo, respectivamente, del emulador de FlashROM. En la sección 4.3 se explica el funcionamiento de estos programas.

B.1 ihxwrite

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "flashemu.h"

short int *flashemu_data;
int do_read = 1;


unsigned int get_value_at(char *l, int width) {
    unsigned int nibble, value;
    int i;

    value = 0;
    for (i = 0; i < width; i++) {
        if ((l[i] > '9') || (l[i] < '0'))
            nibble = 10 + (l[i] - 'A');
         else
            nibble = l[i] - '0';
```

```
        value += nibble << (4 * (width - i - 1));
        }
    return value;
    }

int send_line(char *l, int n, short int *data_array) {
    int num_bytes, i;
    int record_type;
    unsigned char data;
    int checksum;
    int dummy;
    unsigned int address;
    char trash[80];

    /*  record signature  */
    if (l[0] != ':') {
        fprintf(stderr, "%d : Invalid record\n", n);
        return;
        }
    /*  num bytes and starting address  */
    num_bytes = get_value_at(&l[1], 2);
    address = get_value_at(&l[3], 4);
    record_type = get_value_at(&l[7], 2);
    printf("%5d : num_bytes = %2X. address = %4X\n", n, num_bytes, address);
    /*  check record type  */
    if (record_type == 0x01)
        return 0;
    if (record_type != 0x00) {
        fprintf(stderr, "%d : Unknown record type (%2X)\n", n, record_type);
        return 1;
        }
    /*  read data  */
    dummy = num_bytes + record_type + (address & 0x00FF) + (address >> 8);
    printf("\t");
    for (i = 0; i < num_bytes; i++) {
        data = get_value_at(&l[9 + (i * 2)], 2);
        dummy += data;
        printf("[%4X]%2X ", address + i, data);
        flash_emu_write_at(address + i, data);
        data_array[address + i] = data + 1;
        }
    printf("\n");
    /*  read checksum  */
    checksum = get_value_at(&l[9 + (num_bytes * 2)], 2);
    dummy &= 0x00FF;
    dummy = 0x0100 - dummy;
    dummy &= 0x00FF;
```

```
    if (checksum != dummy)
        fprintf(stderr, "%d : Checksum error (readed = %2X, calculated = %2X)\n", n, che
    return 1;
    }

int main(int argc, char *argv[]) {
    char line[256];
    char trash;
    int n = 1;
    int address, test_ok, i;
    FILE *f;

    /*  test command line  */
    for (i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-nr") == 0)
            do_read = 0;
        else {
            /*  open file  */
            f = fopen(argv[i], "rt");
            if (f == NULL) {
                perror("open file");
                return -1;
                }
            break;
            }
        }

    printf("Testing %s\n", (do_read) ? "ENABLED" : "DISABLED");

    /*  alloc flashemu data for later comparing  */
    flashemu_data = (short int*) malloc(65536 * sizeof(short int));
    if (flashemu_data == NULL) {
        perror("allocating flashemu data");
        return -1;
        }

    /*  initializing flash emu  */
    printf("Set flashemu in WRITE mode and press a key");
    fflush(stdout);
    read(0, &trash, 1);

    flash_emu_init();

    /*  writting data  */
    while (fscanf(f, "%[^\n]\n", line) == 1) {
        if (!send_line(line, n++, flashemu_data))
            break;
```

```
        }
    printf("Total: %d lines\n", n - 1);

    /*  close file  */
    fclose(f);

    if (do_read) {
        /*  testing  */
        printf("Set flashemu in READ mode and press a key");
        fflush(stdout);
        read(0, &trash, 1);

        test_ok = 1;
        for (i = 0; i < 65536; i++) {
            if (flashemu_data[i] == 0)
                continue;
            if ((flashemu_data[i] - 1) != flash_emu_read_at(i)) {
                printf("Test FAILED at %X\n", i);
                test_ok = 0;
                }
            }
        if (test_ok)
            printf("Test OK\n");
          else {
            printf("Test WRONG\n");
            while (scanf("%X", &address) == 1)
                printf("==> %X\n", flash_emu_read_at(address));
            }
        }
     else
        printf("Writting OK\n");

    /*  terminate flashemu  */
    flash_emu_done();
    free(flashemu_data);

    printf("Flashemu set to high Z\n");
    return 0;
    }
```

B.2 ihxtest

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include "flashemu.h"


unsigned int get_value_at(char *l, int width) {
    unsigned int nibble, value;
    int i;

    value = 0;
    for (i = 0; i < width; i++) {
        if ((l[i] > '9') || (l[i] < '0'))
            nibble = 10 + (l[i] - 'A');
         else
            nibble = l[i] - '0';
        value += nibble << (4 * (width - i - 1));
        }
    return value;
    }

int test_line(char *l, int n) {
    int num_bytes, i;
    int record_type;
    unsigned char data;
    int checksum;
    int dummy;
    unsigned int address;
    char trash[80];

    /*  record signature  */
    if (l[0] != ':') {
        fprintf(stderr, "%d : Invalid record\n", n);
        return;
        }
    /*  num bytes and starting address  */
    num_bytes = get_value_at(&l[1], 2);
    address = get_value_at(&l[3], 4);
    record_type = get_value_at(&l[7], 2);
    printf("%5d : num_bytes = %2X. address = %4X\n", n, num_bytes, address);
    /*  check record type  */
    if (record_type == 0x01)
        return 0;
    if (record_type != 0x00) {
        fprintf(stderr, "%d : Unknown record type (%2X)\n", n, record_type);
        return 1;
        }
    /*  read data  */
```

```
    dummy = num_bytes + record_type + (address & 0x00FF) + (address >> 8);
    printf("\t");
    for (i = 0; i < num_bytes; i++) {
        data = get_value_at(&l[9 + (i * 2)], 2);
        dummy += data;
        printf("[%4X]%2X ", address + i, data);
        if (flash_emu_read_at(address + i) != data)
            printf("Test FAILED at %4X\n", address + i);
        }
    printf("\n");
    /*  read checksum  */
    checksum = get_value_at(&l[9 + (num_bytes * 2)], 2);
    dummy &= 0x00FF;
    dummy = 0x0100 - dummy;
    dummy &= 0x00FF;
    if (checksum != dummy)
        fprintf(stderr, "%d : Checksum error (readed = %2X, calculated = %2X)\n", n, che
    return 1;
    }

int main(int argc, char *argv[]) {
    char line[256];
    char trash;
    int n = 1;
    int address, test_ok, i;
    FILE *f;

    fprintf(stderr, "start\n");
    /*  open file  */
    f = fopen(argv[1], "rt");
    if (f == NULL) {
        perror("open file");
        return -1;
        }

    /*  initializing flash emu  */
    printf("Set flashemu in READ mode and press a enter");
    fflush(stdout);
    read(0, &trash, 1);
    flash_emu_init();

    /*  writting data  */
    while (fscanf(f, "%[^\n]\n", line) == 1) {
        if (!test_line(line, n++))
            break;
        }
    printf("Total: %d lines\n", n - 1);
```

```
/*  close file  */
fclose(f);

/*  terminating flash emu  */
flash_emu_done();

printf("Flashemu set to high Z\n");
return 0;
}
```

Apéndice C

La estructura del CD

En el CD adjunto a este documento se incluyen todos los códigos fuente, las hojas de datos de los chips utilizados y demás documentación.

- `doc`
  Directorio de documentación.
  - `datasheets`
    Hojas de datos de los chips utilizados en este proyecto (80592, 62256, 29F010, etc).
  - `latex`
    Todo el código fuente de este documento en LaTeX, las figuras en formato `fig` y `eps`, y las fotografías en formato `jpg` y `eps`.
  - `powerpoint`
    La presentación del proyecto en formato Microsoft PowerPoint.
  - `misc`
    Documentación adicional (tutorial de lógica difusa, especificación del estándar EPP/ECP de puerto paralelo y manual original completo del SDCC).
- `src`
  Directorio de códigos fuente.
  - `asm`
    Ejemplos de código fuente en ensamblador para el 80592.
  - `flashemu`
    La librería `flashemu` para acceder a emulador de FlashROM y los programas que hacen uso de ella: `ihxwrite` e `ihxtest`.
  - `microcyc_fuzz`
    El código fuente del software `microcyc_fuzz` (flex, bison y C) con algunos códigos ejemplo de programas escritos en *fuzz* (el código `ft3.fuzz` es el código que se utilizó para el control del puente de grúa).

En cada uno de los directorios que contienen código fuente (`asm`, `flashemu` y `microcyc_fuzz`) existe un fichero `Makefile` que permite compilar cada uno de los programas en Linux. El `Makefile` del directorio `asm` utiliza el ensamblador `asx8051` y el compilador `sdcc` para generar los ejemplos.

En el directorio `latex` existe también un `Makefile` que permite construir este documento en formato PostScript.

Bibliografía

[1] Bernard Odant.
Microcontroladores 8051 y 8052.
Editorioal Paraninfo, 1995.

[2] Bonifacio Martín del Brío y Alfredo Sanz Molina.
Redes Neuronales y Sistemas Borrosos.
Editorial Ra-Ma, 2001.

[3] Alfred V. Aho, Ravi Sethi y Jeffrey D. Ullman.
Compiladores: Principios, Técnicas y Herramientas.
Editorial Addison-Wesley Iberoamericana, 1986.

[4] Francisco Manuel Márquez.
UNIX. Programación avanzada.
Editorial Ra-Ma, 1996.

[5] Philips Semiconductors.
P8xC592 Microcontroller Datasheet.

[6] Philips Semiconductors.
PCA82C250 CAN Controller Interface Datasheet.

[7] Analog Devices.
74573 8-bit 3-state Transparent Latch Datasheet.

[8] Advanced MicroDevices.
Am29F010 Flash Memory Datasheet.

[9] Hitachi Semicontuctors.
HM62256B Static RAM Datasheet.

[10] J. S. R. Jang, C. T. Sun y E. Mizutani.
A Computational Approach to Learning and Machine Intelligence.
Prentice Hall, 1997.

[11] Kevin M. Passino y Stephen Yurkovich.
Fuzzy Control.
Addison-Wesley, 1998.

[12] Li-Xi Wang.
Adaptative Fuzzy Systems and Control.
Prentice Hall, 1994.

Referencias en Internet

- Hojas de datos y fabricantes de chips.
  - `http://www.semiconductors.philips.com`
    Philips Semiconductor (Microcontrolador y transceptor CAN).
  - `http://semiconductor.hitachi.com`
    Hitachi Semicontuctor (SRAM).
  - `http://www.amd.com`
    Advanced MicroDevices (FlashROM).
  - `http://www.embeddedlinks.com/chipdir`
    Chip Directory (74573 y hojas de datos de chips estándar).
- Protocolo CAN.
  - `http://www.algonet.se/~staffann/developer/CAN.htm`
    Tutorial CAN para programación de microcontroladores.
  - `http://www.mjschofield.com`
    Directorio sobre recursos CAN.
- Control borroso:
  - `http://www.aptronix.com/fide/whyfuzzy.htm`
    Introducción al control borroso.
  - `http://www.bytecraft.com/fuzlogic.pdf`
    Manual de Byte Craft Limited Software para la implementación de controladores borrosos en dispositivos pequeños.