Sistemas Embebidos

Practica 5.
Uso de procedimientos o subrutinas
Configuración del Display LCD 1602

Objetivos:

- Que el estudiante configure cooonff.
-

Material

- Software Atmel Studio 7
- Tarjeta protoboard
- Tarjeta Arduino Atmega
- 1 Resistencia de 10KΩ variable (potenciómetro)
- 1 Display LCD 1602A
- Cables de conexión

1. Uso de subrutinas

Una subrutina (procedimiento o función) es un grupo de instrucciones que realizan una tarea específica. Las subrutinas son secciones de código que puede ser reutilizables, es decir, que se guardan una vez en memoria y pueden usarse vareas veces en un mismo programa. Esto permite utilizar la estrategia "divide y vencerás" en el análisis, diseño y construcción de software.

El MCU Atmega 2560 contiene las instrucciones CALL, RCALL, ICALL y EICALL para invocación a subrutinas y la instrucción RET para el regreso de subrutina. La tabla muestra un resumen de las instrucciones para la invocación a subrutinas.

Tabla 1.1 Resumen de las instrucciones de invocación a subrutinas

Instrucción	Pseudocódigo	Nombre	Descripción
CALL k	*(SP) ← PC + 1 SP ← SP - 1 PC ← k	**Llamada a subrutina**	Llamada a subrutina, La dirección de regreso (La dirección de la siguiente instrucción de CALL) es almacenada en la pila y se actualiza el PC con la constante k.
RCALL k	*(SP) ← PC + 1 SP ← SP - 1 PC ← PC + k + 1	**Llamada relativa a subrutina**	Llamada relativa a subrutina, La dirección de regreso (La dirección de la siguiente instrucción de RCALL) es almacenada en la pila y se actualiza el PC a la dirección donde está definida la subrutina (PC + k + 1).
ICALL k	*(SP) ← PC + 1 SP ← SP - 1 PC(15:0) ← Z	**Llamada indirecta a subrutina**	Llamada indirecta a subrutina, La dirección de regreso (La dirección de la siguiente instrucción de ICALL) es almacenada en la pila y se actualiza el PC a la dirección almacenada en el registro índice Z.
EICALL k	*(SP) ← PC + 1 SP ← SP - 1 PC(15:0) ← Z PC(21:16) ← EIND	**Llamada a subrutina en modo indirecto extendido**	Llamada a subrutina en modo indirecto extendido, La dirección de regreso (La dirección de la siguiente instrucción de EICALL) es almacenada en la pila y se actualiza el PC a la dirección almacenada en el registro índice Z junto con el registro EIND (Extended Indirect Register)
RET	PC ← *(SP) SP ← SP + 1	**Regreso de subrutina**	Regreso de subrutina, se extrae de la pila la dirección de regreso de subrutina y es almacenada en el PC

El proceso de invocación a subrutina comienza con la ejecución de una llamada a subrutina con una instrucción CALL, el sistema automáticamente guarda la dirección de regreso en la pila del sistema, después, el contador de programa (PC) se actualiza con la dirección donde se encuentra definida la subrutina, esto hace, que la ejecución continúe en la primera instrucción de la subrutina, después se ejecutan todas las instrucciones definidas en dicho procedimiento, finalmente el regreso de subrutina ocurre cuando se ejecuta la instrucción RET (regreso de subrutina), esto ocasiona que se recupere la dirección de regreso desde la pila del sistema y la asigna al contador de programa (PC), por lo que, la siguiente instrucción a ejecutar sería, la instrucción que le sigue a la instrucción CALL. Ver figura 1.1.

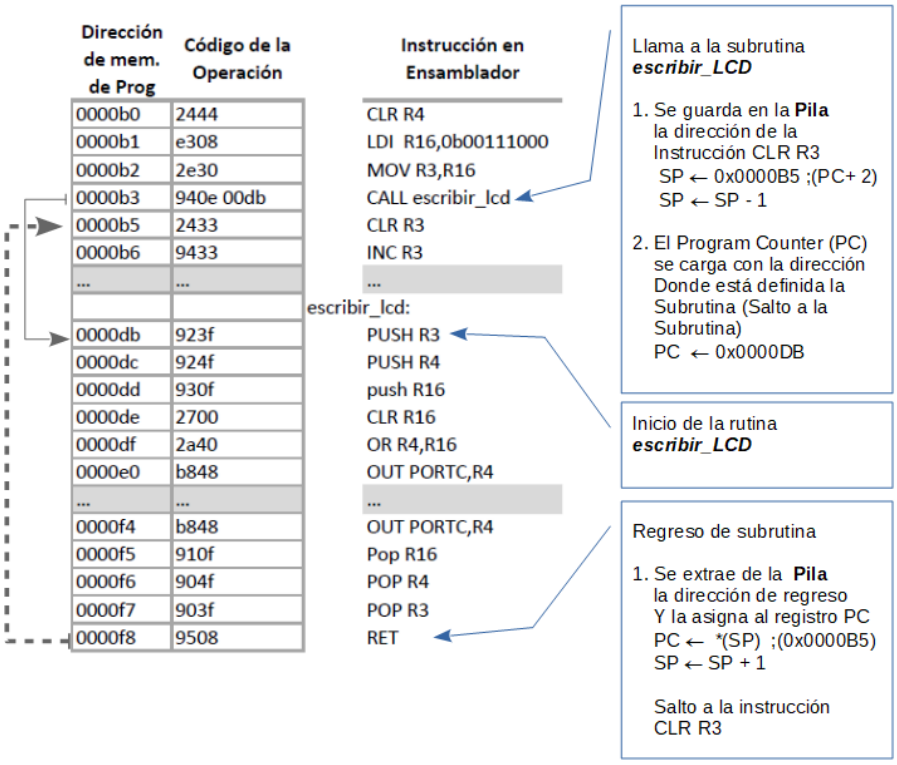


Figura 1.1. Proceso de invocación a subrutina en el MCU ATmega 2560.

2. La pila del sistema

El MCU Atmega 2560 contiene el registro SP (*Stack Pointer*), que permite realizar una estructura de datos LIFO (*Last-In, First-Out*) nivel de hardware; también llamada Pila del Sistema: es un espacio de memoria RAM donde se guardan temporalmente datos, direcciones de memoria, parámetros de entrada o variables locales (ver figura 2.1). El MCU tiene instrucciones propias para agregar (PUSH) o extraer (POP) datos hacia o desde la pila; también incluye, instrucciones que usan la pila de forma automática, como la instrucción CALL que agrega a la pila la dirección de regreso de la subrutina, y la instrucción RET que extrae la dirección de regreso de subrutina de la pila. Así mismo, la lógica para la invocación a rutinas de interrupción y regreso de interrupción, utilizan de forma automática la pila del sistema. Por estas razones, es importante que la pila del sistema se inicialice, antes del uso de interrupciones o procedimiento, a una localidad de memoria RAM con suficiente espacio para cubrir las necesidades de almacenamiento del programa.

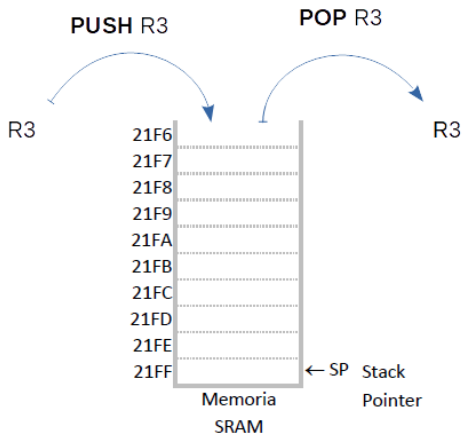


Figura 2.1. Proceso de agregar y extraer un valor a la pila del sistema.

El MCU contiene los registros SPH y SPL de 8-bits que en conjunto forman el apuntador a la pila del sistema (SP, *Stack Pointer*), ver figura 2.1. El registro SPL guarda la parte baja de la dirección de memoria RAM donde se encuentra el tope de la pila y el registro SPH guarda la parte alta de dicha dirección. Estos registros se encuentran mapeado en el espacio de memoria de los registros de I/O, por lo que se puede acceder a ellos con instrucciones con direccionamiento relativo como son IN y OUT.

Bit	15	14	13	12	11	10	9	8	
0x3E (0x5E)	SP15	SP14	SP13	SP12	SP11	SP10	SP9	SP8	SPH
0x3D (0x5D)	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	SPL
	7	6	5	4	3	2	1	0	
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	1	
	1	1	1	1	1	1	1	1	

Figura 2.2. Registros que forman el apuntador a la pila SP

El registro SP debe ser inicializado al comienzo del programa, con la dirección de memoria donde termina el área definida como pila del sistema, por lo regular se inicializa con la última dirección de la memoria SRAM interna (para el caso del microcontrolador Atmega2560, la última casilla de memoria RAM es 0x21FF). Se puede guardar el contenido de un registro a la pila, mediante la instrucción PUSH, lo que ocasionará que el dato del registro se guarde en la memoria SRAM y el registro apuntador a la pila SP, se decremente en una unidad; también se hace uso de la pila cuando se invoca una subrutina (con la instrucción CALL, RCALL e ICALL) o se invoca interrupción, el sistema guarda automáticamente la dirección de regreso, a la pila (dirección formada por 3 bytes para el microcontrolador Atmega2560) y el registro SP es decrementado en tres. Para extraer un dato de la pila se emplea la instrucción POP, que lee el dato que está en el tope de la pila y lo guarda en un registro, esta instrucción incrementa a la pila en uno; las instrucciones de regreso de subrutina (RET) y de regreso de interrupción (RETI) guardan el contenido del tope de la pila al registro contador de programa PC e incrementan al registro SP en 3.

3. Declaración de subrutinas

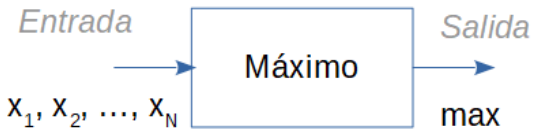
En el desarrollo de software de proyectos pequeños, por lo regular se emplea la estrategia de diseño modular donde se propone la descomposición del proyecto en módulos que resuelven una tarea específica, cada módulo pueden contener una o más funciones o procedimientos orientados a resolver la tarea propia del módulo. Una de las ventajas de usar esta estrategia es que el problema se puede dividir en partes, además, si es el caso de un buen diseño de las subrutinas, la posibilidad de reusar el código, por tanto, la posibilidad de crear bibliotecas de funciones que se pueden reusar en otros proyectos. La arquitectura del MCU Atmega 2560 tienen lo necesario para la realización de módulos, como instrucciones de invocación a subrutinas y manejo de una pila a nivel de hardware.

En la elaboración de subrutinas (procedimientos o funciones), es necesario especificar los parámetros que recibe la subrutina y los datos que regresa. En lenguaje ensamblador, los parámetros de entrada pueden ser especificados en los registros del sistema, así mismo, los datos de regreso; también se puede emplear la pila del sistema, para guardar los argumentos de entrada de la subrutina y de la misma forma el valor de regreso.

Por ejemplo. Se desea elaborar un programa en lenguaje ensamblador que determine el valor máximo de una secuencia de N números enteros con signo de 8-bits.

Análisis

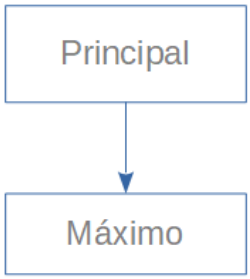
Entrada: {x_1, x_2, …, x_N}: N números entero con signo de 8-bits
Salida: mayor: Entero con signo de 8-bits



Restricciones:

x_1, x_2, …,x_N, max $\in$ {-128, -127, …, +126, +127}
N $\in$ {1, 2, …,127}

El proyecto se puede descomponer en dos subrutinas: Principal y máximo



El diseño de los algoritmos para cada uno de los módulos, se tienen en Algoritmo 1 y Algoritmo 2

Algoritmo 1: Módulo Principal

Entrada: $\{x_1, x_2, \cdots, x_N\}$: Arreglo de números enteros con signo de 8 bits
Salida: max: Valor máximo, entero con signo de 8-bits
inicio
  Entero: N, max
  Apuntador: dirArreglo
  $dirArreglo \leftarrow$ Obtener dirección del arreglo
  $N \leftarrow$ Tamaño del arreglo
  $max \leftarrow$ **Maximo**(dirArreglo, N) *//llamado a subrutina*
fin

Algoritmo 2: Módulo Máximo

Entrada: $dirArreglo$: Dirección de memoria Flash donde se guardó el arreglo, enteros sin signo de 16 bits
  N: Número de elementos en el arreglo, entero con signo de 8-bits
Salida: max: elemento máximo del arreglo, entero con signo de 8-bits
inicio
  Entero: i, dato, max
  Apuntador: pArray
  si $N \leq 0$ **entonces**
    Regresa 0
  fin
  $pArray \leftarrow dirArreglo$
  $i \leftarrow 1$
  $max \leftarrow pArray[0]$
  mientras $i < N$ **hacer**
    $dato \leftarrow pArray[i]$
    si $max \leq dato$ **entonces**
      $max \leftarrow dato$
    fin
    $i \leftarrow i + 1$
  fin
  Regresa max
fin

El siguiente paso a seguir, es la codificación de los algoritmos a lenguaje ensamblador. En este apartado se debe tener un poco de cuidado a la transcripción de las instrucciones en lenguaje ensamblador, esto debido a que tenemos recursos de hardware muy específicos e instrucciones en ensamblador particulares, además que se requiere que el programa siga el paradigma de programación modular, por esta razón, se ha definido algunos pasos que deben realizarse en el proceso de codificación en lenguaje ensamblador.

1. El programa principal debe inicializar la pila del sistema con la última dirección de memoria SRAM, cuando utilice subrutinas y/o interrupciones.

2. Para el programa principal y cada subrutina, es necesario definir cuales registros del CPU representarán los parámetros de entrada y salida y/o variables locales del algoritmo. Si es necesario, reservar casillas de memoria RAM, en la pila del sistema.

3. Antes de invocar a una subrutina, se debe inicializar los parámetros de entrada con los valores de los argumentos, siguiendo la convención establecida en el paso previo.

4. Cuando se define una subrutina, se debe guardar el estado de todos los registros utilizados por la subrutina en la pila del sistema, al inicio de cada subrutina; y al final, se debe reestablecer el estado original de dichos registros previo al regreso de la subrutina.

5. Es recomendable codificar el cuerpo de los algoritmos, siguiendo el paradigma estructurado, es decir, la realización de las estructuras de control en lenguaje ensamblador, evitando saltos que rompen dichas estructuras.

6. Reservar espacios en memoria SRAM para guardar variables globales.

7. Reservar espacios en memoria FLASH para guardar constantes globales

Siguiendo con el ejemplo, para codificar de los algoritmos de debe definir lo siguiente.

1. Inicializar la pila del sistema con la dirección de la última casilla de la memoria SRAM, en la función principal.

   En esta parte es conveniente hacer uso de la directiva `.EQU` para definir una etiqueta que represente la dirección de la última casilla de memoria SRAM. El siguiente código realiza la inicialización.

```
1
2       .EQU FIN_RAM = 0x21FF
3
4   PRINCIPAL:
5       ; 1. Se inicializa la pila con el última localidad de memoria SRAM
6
7       LDI R16, LOW(FIN_RAM)
8       OUT SPL, R16
9       LDI R16, HIGH(FIN_RAM)
10      OUT SPH, R16
11
```

2. Para el programa principal y cada subrutina, se debe definir cuales registros del CPU representarán los parámetros de entrada y salida y/o variables locales del algoritmo. Si es necesario, reservar casillas de memoria RAM, en la pila del sistema.

   La tabla 3.1 muestra las variables locales usadas por el programa principal.

Tabla 3.1 Definición de registros para representar variables locales en la función principal

Nombre de variable	Tipo de Variable	Tipo de datos	Recurso asignado
dirArreglos	Variable Local	Apuntador de memoria de programa. Enteros sin signo de 16-bits	Registro Z
N	Variable Local	Entero con signo de 8-bits	Registro R16
max	Variable Local	Entero con signo de 8-bits	Registro R0

El programa principal invoca a la subrutina Máximo(), con el siguiente prototipo

```
char Maximo(char *dirArreglo, char N);
```

La tabla 3.2 muestra los registros utilizados como variables locales y los parámetros de entrada y salida en la función Máximo

Tabla 3.2 Definición de registros para representar variables locales y parámetros de entrada y salida, en la función Maximo.

Nombre de variable	Tipo de variable	Tipo de datos	Recurso asignado
pArreglos	Entrada	Apuntador de memoria de programa. Enteros sin signo de 16-bits	Registro Z
N	Entrada	Entero con signo de 8-bits	Registro R16
i	Variable local	Entero con signo de 8-bits	Registro R0
dato	Variable local	Entero con signo de 8-bits	Registro R1
max	Variable local	Entero con signo de 8-bits	Registro R2
mayor	Salida	Entero con signo de 8-bits	Registro R16

3. Antes de invocar a una subrutina, se debe inicializar los parámetros de entrada con los valores de los argumentos, siguiendo la convención establecida en el paso previo.

El siguiente código de la función principal, invoca a la subrutina Máximo(), primero inicializa los argumentos de entrada y al final ejecuta la instrucción de invocación.

```
11
12      ; 2. Lectura de la dirección del arreglo de la memoria FLash
13      LDI ZL, LOW(dirArray<<1)
14      LDI ZH, HIGH(dirArray<<1)
15
16      ; 3. Lectura del tamaño del arreglo
17      LDI R16, 5          ;N <-- 5
18
19      ; 4. Invocación a la subrutina Maximo()
20      ;    El valor de regreso se guarda en R16
21      CALL Maximo
```

4. Cuando se define una subrutina, se debe guardar el estado de todos los registros utilizados por la subrutina en la pila del sistema, al inicio de cada subrutina; y al final, se debe reestablecer el estado original de dichos registros previo al regreso de la subrutina.

   La subrutina Máximo(), define tres las variables locales: `i, dato` y `max`; las cuales son representadas por los registros `R0, R1` y `R2`, respectivamente. En el siguiente código, se muestra la manera de guardar el estado de los registros .en al pila del sistema.

```
1  /* Subrutinas.asm  */
2
3  /* Subrutina: Maximo
4   * Entrada: Z: dirArreglo, Dirección de memoria de programa donde está guardado el arreglo
5   *          R16: N, Número de elementos del arreglo
6   * Salida:  R16: max, valor máximo del arreglo
7   */
8      .CSEG
9  Maximo:
10         ; 1. Se guarda el estado de los registros usados en la pila
11         PUSH  R0    ; RO := i
12         PUSH  R1    ; R1 := dato
13         PUSH  R2    ; R2 := maximo
14
```

   El siguiente código restaura el contenido original de los registros antes del regreso a subrutina.

```
44
45 REGRESO_SUBRUTINA:
46         POP R2      ;Restaura el estado de los registros
47         POP R1
48         POP R0
49         RET         ;Regreso de subrutina
50
```

5. Es recomendable codificar el cuerpo de los algoritmos, siguiendo el paradigma estructurado, es decir, la realización de las estructuras de control en lenguaje ensamblador, evitando saltos que rompen dichas estructuras.

   El siguiente código muestra el código completo de la subrutina Maximo().

```
1  /* Subrutinas.asm  */
2
3  /* Subrutina: Maximo
4   * Entrada: Z: dirArreglo, Dirección de memoria de programa donde está guardado el arreglo
5   *          R16: N, Número de elementos del arreglo
6   * Salida:  R16: max, valor máximo del arreglo
7   */
8      .CSEG
9  Maximo:
10         ; 1. Se guarda el estado de los registros usados en la pila
11         PUSH  R0    ; RO := i
12         PUSH  R1    ; R1 := dato
13         PUSH  R2    ; R2 := maximo
14
15         CLR R0      ; i<-- 0
16
17         ; 2. Si N <= 0 entonces regresa la subrutina
18         CP R0, R16      ;Si N > 0 entonces
19         BRLT  FIN_SI    ;Salta al FIN_SI_1
20         CLR R16
21         RJMP  REGRESO_SUBRUTINA ; Regreso de subrutina
22
23 FIN_SI:
24         ; 3. Se inicializan las variables
25         INC R0          ; i<--1
26         LPM R2, Z       ; max <--pArray[0]
27
28         ; 4. Mientras i < N hacer
29 INICIO_WHILE:
30         CP R0, R16
31         BRGE FIN_WHILE  ;Salta al fin del while cuando ya no se cumple la condición
32
33         ; 5. dato <-- *pArray
34         LPM R1, Z+      ; pArray <-- pArray + 1
35
36         ; 6. Si max < dato entonces
37         CP R2, R1   ; Compara max >= dato si cumple salta al FIN_SI2
38         BRGE FIN_SI2
39         MOV R2, R1  ;max <-- dato
40 FIN_SI2:
41         INC R0      ; i <-- i + 1
42         RJMP INICIO_WHILE
43 FIN_WHILE:
44
45 REGRESO_SUBRUTINA:
46         POP R2      ;Restaura el estado de los registros
47         POP R1
48         POP R0
49         RET         ;Regreso de subrutina
50
```

El siguiente código muestra la codificación de la función principal.

```
1
2       .EQU FIN_RAM = 0x21FF
3
4   PRINCIPAL:
5       ; 1. Se inicializa la pila con el última localidad de memoria SRAM
6
7       LDI R16, LOW(FIN_RAM)
8       OUT SPL, R16
9       LDI R16, HIGH(FIN_RAM)
10      OUT SPH, R16
11
12      ; 2. Lectura de la dirección del arreglo de la memoria FLash
13      LDI ZL, LOW(dirArray<<1)
14      LDI ZH, HIGH(dirArray<<1)
15
16      ; 3. Lectura del tamaño del arreglo
17      LDI R16, 5          ;N <-- 5
18
19      ; 4. Invocación a la subrutina Maximo()
20      ;    El valor de regreso se guarda en R16
21      CALL Maximo
22
23      ; 5. El resultado se guarda en el registro R0
24
25      MOV R0, R16
26
27  FIN:
28      RJMP FIN
29
30  ; Incluye la definición de la subrutina
31
32      .INCLUDE "Subrutinas.asm"
33
34  ;Definición de la sección de datos en la memoria de programa
35      .CSEG
36  dirArray:   .DB  3,2,6,-1,6, 0      ;arreglo = {3, 2 ,9 , -1 ,6}
37
```

Observar que al final se incluye el archivo de las subrutinas y la declaración del arreglo como una tabla de valores constantes.

4. Desarrollo de proyectos de proyectos que combinan rutinas escritas en lenguaje C y lenguaje ensamblador.

Muchos de los proyectos en sistemas embebidos, combinan código escrito en lenguaje ensamblador y lenguaje C. Principalmente, se deja a las rutinas en ensamblador aquellas tareas que requieren un mejor aprovechamiento de las características de hardware, aquellas tareas que requieren tener un control eficiente de memoria y tiempo de ejecución. Así mismo, se deja a las rutinas escritas en lenguaje C, las tareas que no dependen en mucho del tiempo de ejecución ni tampoco del espacio requerido por el programa.

Existen muchas ventajas y desventajas al usar cualquiera de los lenguajes tanto en ensamblador como en lenguaje C. ¿Cuál usar? Dependerá del tipo de proyecto se desee implementar, Una de las ventajas de lenguaje C, es su nivel de abstracción, que no requiere mucho conocimiento del hardware, sin embargo, el compilador genera código objeto no tan eficiente como lo podría hacer un código bien realizado en lenguaje ensamblador; así mismo, hay la posibilidad, de que el tamaño del programa objeto generado por compilador, supere la capacidad de almacenamiento de la memoria de programa del MCU por lo que no se podría implementar bajo esta arquitectura.

4.1. Diferencias entre el ensamblador AVR Assambler y AVR/GNU GCC Assembler.

En el entorno de desarrollo (IDE) Atmel Studio, se pueden crear dos tipos de proyectos para las arquitecturas AVR de 8-bits: el primero utilizando la herramienta de generación de código *AVR Assembler* (avrasm2), en este tipo de proyectos, sólo se pueden agregar archivos en lenguaje ensamblador con extensión *.asm, no se pueden incluir archivos de otro lenguaje como C/C++; su herramienta de generación de código, no genera archivos objeto y por tanto, no tiene un ligador (*linker*), si no que directamente traduce los archivos fuente a un archivo ejecutable *.hex. La segunda forma de crear un proyecto es con la herramienta de generación de código AVR/GNU C/C++ Compiler (avr-gcc, avr-g++), este tipo de proyectos, pueden estar formado por archivos fuente escritos en lenguaje C/C++ (*.c, *.cpp) y archivos en lenguaje ensamblador (*.s) con sintaxis de directivas del avr-gcc Assembler. Es necesario aclarar, que los archivos fuente escritos para el ensamblador AVR Assembler (archivos con extensión *.asm) no son compatibles en su totalidad con el compilador avr-gcc, de la misma forma, archivos en lenguaje ensamblador para el avr-gcc Assembler (archivos con extensión *.s), no son compatibles en su totalidad con el ensamblador AVR Assembly. La tabla 4.1 muestra algunas directivas del ensamblador avr-gcc assembler que son equivalentes al ensamblador AVR Assembler.

Tabla 4.1. Directivas de los ensambladores AVR/GNU GCC Assembler y AVR Assembler.

Directivas del AVR/GNU GCC Assembler	Directivas del Atmel AVR Assembler	Descripción
.section .text	**.CSEG**	Especifica el inicio de la sección de código, el código escrito posterior a la directiva se guardará en la memoria de programa.
.section .data	**.DSEG**	Especifica el inicio de la sección de datos inicializados, el código seguido a la directiva será almacenado en la memoria SRAM. La directiva .DREG, del ensamblador AVR, no permite especificar datos inicializados en memoria SRAM, solamente permite inicializar los datos en la sección de memoria de programa.
.section .bss	**.DSEG**	Especifica el inicio de la sección de datos no inicializados, el código seguido a la directiva será almacenado en la memoria SRAM.
.org *dir*	**.ORG *dir***	Establece la dirección donde se ubicará el código (si se establece en la sección de programa) o las variables (si se emplea en la sección de datos). Esta directiva emplea un contador de memoria para especificar la dirección de memoria y solo se puede avanzar la dirección (no se puede retroceder en la dirección).

.include *"arch"*	**.INCLUDE *"arch"***	Directiva que le indica al ensamblador que continúe en el archivo especificado he incluya su código como parte del programa, al encontrar el fin del archivo, el ensamblador regresa y continua en el archivo original.
-	**.DEF alias=Rn**	Establece un sinónimo a un registro. En el AVR/GNU GCC Assembler no tiene una directiva equivalente
.equ *etiq, expres*	**.EQU *etiq=expres***	Establece que una etiqueta represente una expresión. No se puede redefinir la etiqueta
.set *etiq, expres*	**.SET *etiq=expres***	Establece que una etiqueta represente una expresión. Se puede redefinir la etiqueta a lo largo del programa.
.extern	**-**	Esta directiva es incluida por compatibilidad con otros lenguajes, establecer que una etiqueta está definida externamente.
.global	**-**	Establece una etiqueta como global, para que sea visible en otras partes del proyecto o solución.
.byte *cte*	**.DB *cte***	Reserva una constante de un byte o una lista de constantes de bytes en memoria de programa o en memoria EEPROM. En el AVR/GNU GCC Assembler tiene la directiva *.byte* que cumple con la misma funcionalidad si se define en la sección de programa con la directiva *.text*.
.hword *cte*	**.DW *cte***	Reserva una constante de 16bits o una lista de constantes de 16-bits en memoria de programa o en memoria EEPROM. En el AVR/GNU GCC Assembler tiene la directiva *.hword* que cumple con la misma funcionalidad si se define en la sección de programa con la directiva *.text*.
.byte *expres*	**.BYTE *expres***	Reserva un byte o una lista de bytes en memoria de datos SRAM y lo inicializa con el resultado de la expresión dada. En el AVR/GNU GCC se debe definir define en la sección de datos con la directiva *.data* o *.bss*
.hword *expres*	**-**	Reserva memoria para guardar un valor de 16-bits o una lista de valores de 16-bits, en la sección de memoria seleccionada.
.ascii *"cadena"*	**-**	Reserva memoria para guardar una o más cadenas, la cadena no finalizan con el carácter fin de cadena ('\0'), en la sección de memoria seleccionada.
.asciz	**-**	Reserva memoria para guardar una o más cadenas, las cadenas finalizan con el carácter fin de cadena ('\0'), en la sección de memoria seleccionada.

El ensamblador *AVR/GNU GCC Assembler* también define funciones de ayuda para evaluar expresiones en tiempo de ensamblador, la tabla 4.2 muestra las funciones equivalentes para ambos ensambladores.

Tabla 4.2. Funciones de ayuda soportadas por AVR/GNU GCC Assembler y AVR Assembler.

AVR/GNU GCC Assembler	Atmel AVR Assembler	Descripción
lo8(*expres***)**	**LOW(***expres***)**	Obtiene el byte menos significativo de la expresión (bit7 al bit0).
hi8(*expres***)**	**HIGH(***expres***)**	Obtiene el segundo byte menos significativo de la expresión (bit15 al bit8).
hlo8(*expres***)**	**BYTE3(***expres***)**	Obtiene el tercer byte de la expresión (bit23 al bit16).
hhi8(*expres***)**	**BYTE4(***expres***)**	Obtiene el cuarto byte de la expresión (del bit31 al bit24).
pm_lo8(*dirMem***)**	**-**	Obtiene el byte menos significativo (bit7 al bit0) de una dirección de memoria de programa o de datos.
pm_hi8(*dirMem***)**	**-**	Obtiene el segundo byte menos significativo (bit15 al bit8) de una dirección de memoria de programa o de datos.
pm_hh8(*dirMem***)**	**-**	Obtiene el tercer byte (bit23 al bit16) de una dirección de memoria de programa o de datos.

4.2. Elaboración de proyectos con código en lenguaje C y ensamblador.

Las herramientas de generación de código AVR/GNU incluyen las herramientas para compilar, ensamblar, deputar y ligar código escrito en lenguaje C, C++ y ensamblador. El compilador como el ensamblador crear archivos objeto que posteriormente, el ligador los reunir para formar un solo archivo ejecutable. De esta forma, las herramientas de generación de código incluyen en sus especificaciones, opciones de compatibilidad entre los distintos lenguajes.

Cuando se elabora un proyecto exclusivamente en lenguaje ensamblador, se deben establecer una convención de donde se guardan los parámetros de entrada y los datos de regreso, en todas las funciones. Lo recomendable, es seguir la misma convención que sigue el compilador en lenguaje C.

Si el proyecto incluye código en lenguaje C, junto con código en lenguaje en ensamblador, es necesario seguir la convención establecida por el compilador de C, para ello es necesario contestar las siguientes interrogantes:

- ✓ ¿Cómo una rutina en lenguaje ensamblador puede ser visible al compilador de C, de tal manera que la rutina en ensamblador pueda ser invocada desde alguna parte del código en lenguaje C?
- ✓ ¿Cómo una función laborada en lenguaje C, puede ser visible ante el ensamblador, para que esta función en C sea invocada desde alguna parte del código en lenguaje ensamblador?
- ✓ ¿Qué convención tiene el Compilador de C para el paso de parámetros de entrada a una función o subrutina, así como, el o los datos de regreso?

- ✓ ¿Cómo se puede definir las variables globales tanto en ensamblador como en lenguaje C para que se puedan usar de manera indistinta en código C o ensamblador?

4.2.1. Visibilidad de rutinas o funciones entre lenguaje en C y ensamblador

Para que una rutina escrita en lenguaje ensamblador pueda ser visible desde un programa escrito en lenguaje C, se debe realizar dos acciones: la primera, en el código ensamblador se debe definir a la subrutina como global con el uso de la directiva `.global`, la segunda acción es declarar el prototipo de la función con parámetros de entrada y salida como externa, usando el modificador `extern`, ya sea en un archivo de cabecera (*.h) o previo a su invocación en el archivo en lenguaje C.

De la misma forma, si se requiere visualizar una función escrita en lenguaje C en un código escrito en ensamblador, es necesario definir al nombre de la función como externa usando a la directiva `.extern` en el código en ensamblador.

Las variables globales son almacenadas en memoria de datos en la SRAM y pueden ser visibles desde un archivo en lenguaje C, si se define como global en el archivo ensamblador con la directiva `.global` y se declara como externa con el modificador `extern` en el archivo en lenguaje C. Similarmente, si se define una variable global en un archivo en lenguaje C, para poder visualizar su identificador, es necesario especificar con la directiva `.extern` en el archivo ensamblador donde se usa.

4.2.2. Convención para el uso de los registros del CPU al invocar una subrutina/función

El Compilador AVR/GNU emplea los registros del CPU para diversas responsabilidades en la invocación a funciones, si se desea incluir funciones tanto en C como en ensamblador, es necesario conocer como el compilador usa estos registros. La tabla muestra el uso de registro por el compilador.

Tabla 4.3. Convención del uso de los registros por el compilador.

Registro(s)	Descripción	Código en ensamblador que invoca a funciones en lenguaje C	Código en lenguaje C que invoca rutinas en lenguaje ensamblador
R0	Temporal	Es un registro temporal y puede ser modificado por la función en lenguaje C, por tanto, si el registro contiene un dato, este debe ser guardado y restaurado antes y después de llamar la función.	Este registro es temporal debe ser salvado antes de ser usado y restaurado al terminar.
R1	Registro cero	El compilador considera que este registro siempre debe ser cero, por lo tanto, antes de invocar a una función en lenguaje C, es necesario establecerlo a cero.	Si la rutina usa este registro, debe asegurar que antes de regresar de la subrutina, este registro debe tener el valor de cero.
R2-R17, R28, r29	Registros salvados por la función/subrutina	Estos registros deben preservar su estado al invocar una función en lenguaje C. Por lo que no	Si una rutina usa estos registros, debe salvar su contenido antes de utilizarlos y restaurarlos antes del

		requieres que el código en ensamblador guarde su estado.	regreso a subrutina.
R18-R27, R30, R31	Registros usados libremente por la función/subrutina	Estos registros pueden ser usados libremente dentro de una rutina/función, por lo que, si el código en lenguaje ensamblador contiene algún dato en estos registros, debe ser salvado/restaurado antes/después de la invocación a la función.	La rutina en lenguaje ensamblador, puede usar libremente estos registros, sin necesidad de guardar su estado.

4.2.3.Paso de parámetros a una subrutina/función

El compilador utiliza los registros del CPU para el paso de parámetros al invocar una subrutina. Si la lista de argumentos es fija, entonces el primer argumento (de izquierda a derecha) es asignado al registro R24 (si el argumento es de 16 bits, se emplea los registros R25:R24), el segundo argumento es asignado al registro R22 (para 16bits R23:R22), el tercer argumento es asignado al registro R20 (para 16 bits R21:R20) y así sucesivamente, hasta el argumento noveno que se guardará en el registro R8 (para 16 bits R9:R8); argumentos adicionales son pasador en la pila del sistema. Cuando se tiene una lista de argumento variable, todos son guardados en la pila en el orden de derecha a izquierda, los argumentos de tipo char se guardan en 2 bytes, se desperdicia el byte más significativo.

El valor de regreso puede ser de hasta 64 bits y debe colocarse en los registros R25 hasta R18 en el orden que marca la Tabla 4.4. Es necesario recalcar que el orden de los registros no es consecutivo, por ejemplo, el byte0 se guarde en el registro R24, el byte1 en el registro R25 y el byte2 en el registro R22, este orden es acorde al almacenamiento de datos de 16-bits.

Tabla 4.4. Distribución de los bytes de regreso en los registros del CPU.

Registro	R19	R18	R21	R20	R23	R22	R25	R24
Orden del Byte	Byte7	Byte6	Byte5	Byte4	Byte3	Byte2	Byte1	Byte0

Espacios de memoria definidos en GNU GCC

Por defecto el compilador GNU GCC asigna todos los datos al espacio de memoria RAM, inclusive datos de sólo lectura (constantes). Los dispositivos AVR incluye, adicional a la memoria SRAM, el espacio de memoria de datos (flash), en donde se puede almacenar datos constantes y para poderlos leer se emplea las instrucciones LPM o ELPM.

El compilador emplea el modificador `__flash` para determinar que las constantes definidas se almacenarán en la sección de memoria de programa (flash). Los datos almacenados en memoria flash deben ser leídos por la instrucción LPM y los apuntadores a esa dirección tienen una longitud de 16-bits.

Por ejemplo, el siguiente código define un arreglo de tamaño 5 y este se almacena en la memoria flash, también se define a la función SumarArreglo() que recibe como parámetro de entrada un apuntador de un arreglo guardado en memoria flash y el número de elementos, la función regresa la suma de los elementos del arreglo.

```
9
10  /* Declara un arreglo y lo guarda en memoria flash*/
11  const __flash  int datosArray[5] = {10,20,30,40,50};
12
13
14  /*Función SumarArreglo, el arreglo es almacenado en memoria flash*/
15 int SumarArreglo(const  __flash int *pDatos, int tam)
16  {
17      int sum = 0, i=0;
18      while(i<tam)
19      {
20          sum = sum + pDatos[i];
21          i = i + 1;
22      }
23      return sum;
24  }
25
```

4.3. Creación de un proyecto que utiliza lenguaje ensamblador y lenguaje C

Para ejemplificar los pasos a seguir al elaborar un proyecto que incluya código en lenguaje C y lenguaje ensamblador, se mostrará como elaborar un proyecto que encienda y apague un led conectado al bit 0 del puerto A, cada medio segundo, el programa no activa la resistencia pull-up interna. La figura 4.1, muestra el diagrama electrónico del proyecto.

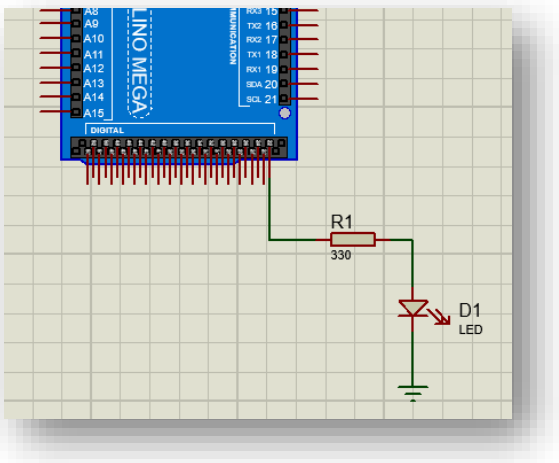
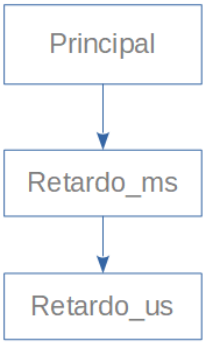


Figura 4.1 Conexión de un led pin0 del puerto A

Por cuestiones de simplicidad, el proyecto se puede descompones en los siguientes módulos.



Descripción de los módulos

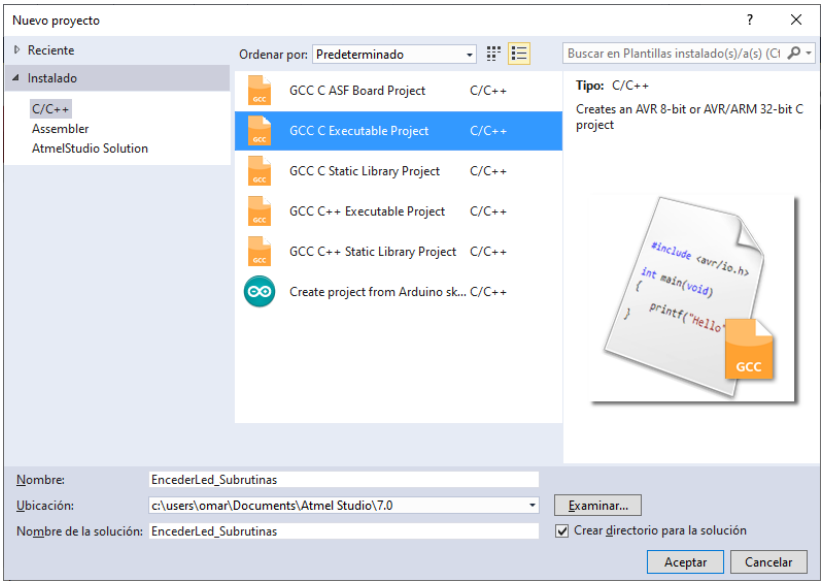
- Principal: Función principal que realiza el control del encendido y apagado del Led.
- Retardo_ms: Esta subrutina realizará un retardo de milisegundos, su prototipo es:

  ```
  void Retardo_ms(unsigned short ms);
  ```

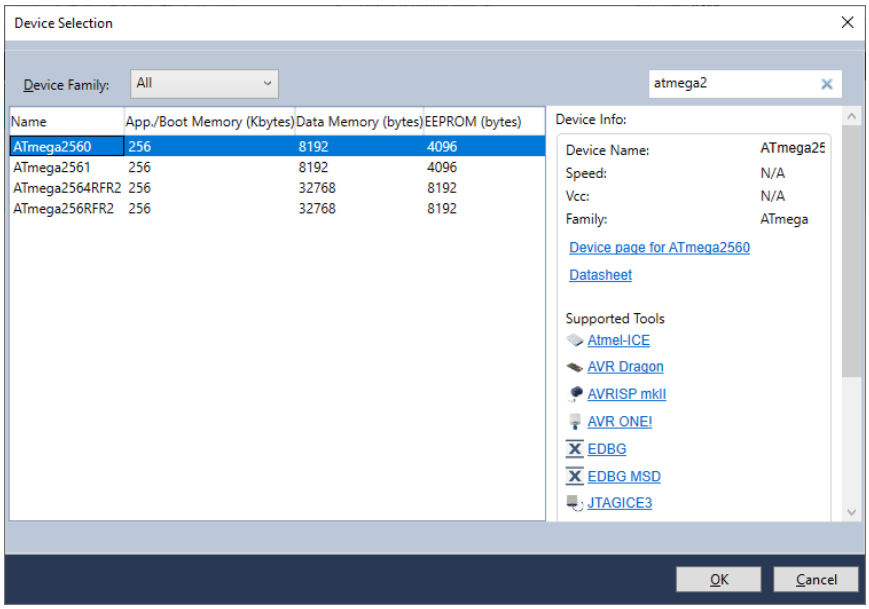
- Retardo_us: Rutina que realiza un retardo de microsegundos, su prototipo es:

  ```
  void Retardo_us(unsigned short us);
  ```

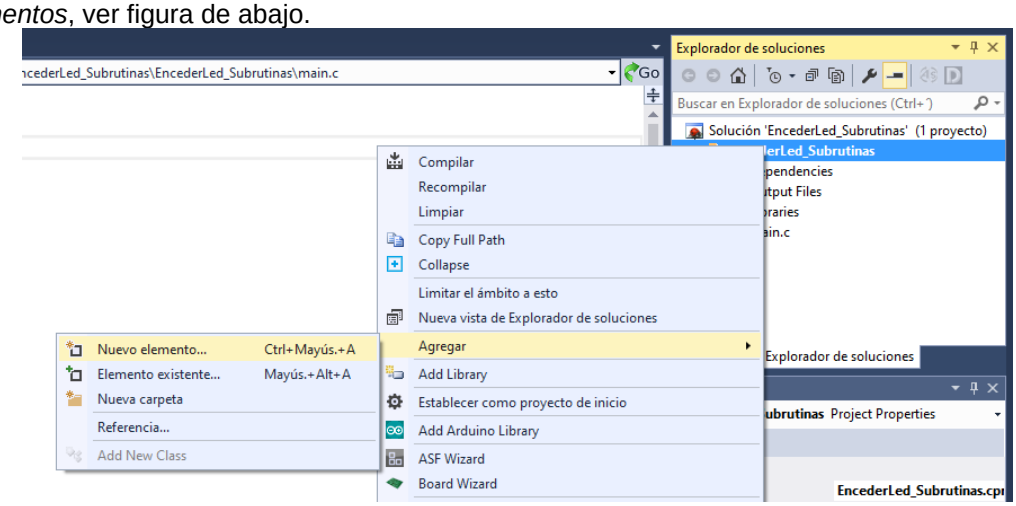
Crear un nuevo proyecto para C/C++



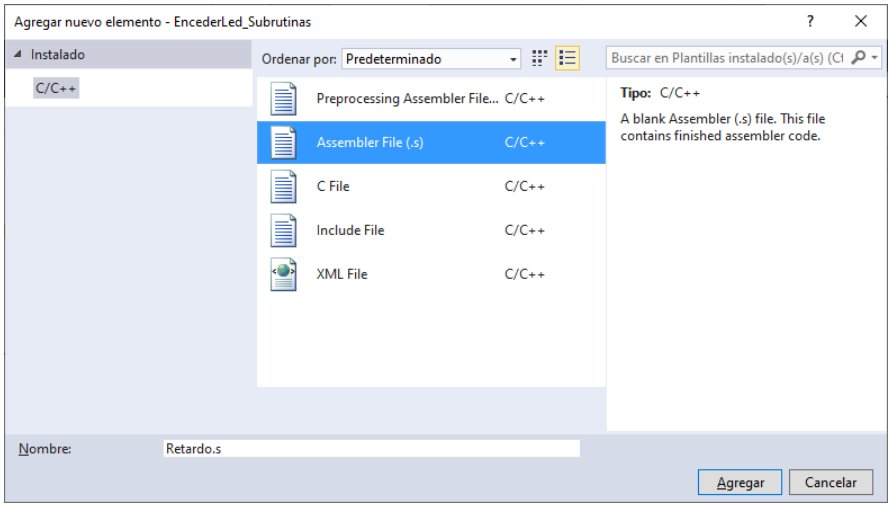
En la selección de dispositivos se elige ATmega2560



A continuación, se agrega un archivo en lenguaje ensamblador con nombre `Retardos.s`. Se hace clic sobre el nombre del proyecto, en el menú contextual elige *Agregar* y después *Nuevo Elementos*, ver figura de abajo.



En el cuadro de diálogos *Agregar Nuevo elemento*, se selecciona la opción *Assembler File (.s)* y se escribe el nombre del archivo como `Retardos.s`



En el archivo se escribe las rutinas en lenguaje ensamblador usando las directivas para el ensamblador AVR/GNU GCC, también se debe seguir las convenciones que usa el compilador para el paso de parámetros a una subrutina y el uso de registros del CPU en subrutinas.

La primera rutina, se ha definido su prototipo como:

```
void Retardo_us(unsigned short us);
```

El parámetro de entrada `us` es un entero 16-bits sin signo, y se almacenará el valor de su argumento en los registros R25:R24. Se valor representa la cantidad de microsegundos (μs) de retardo.

La segunda rutina, se define como:

```
void Retardo_ms(unsigned short ms);
```

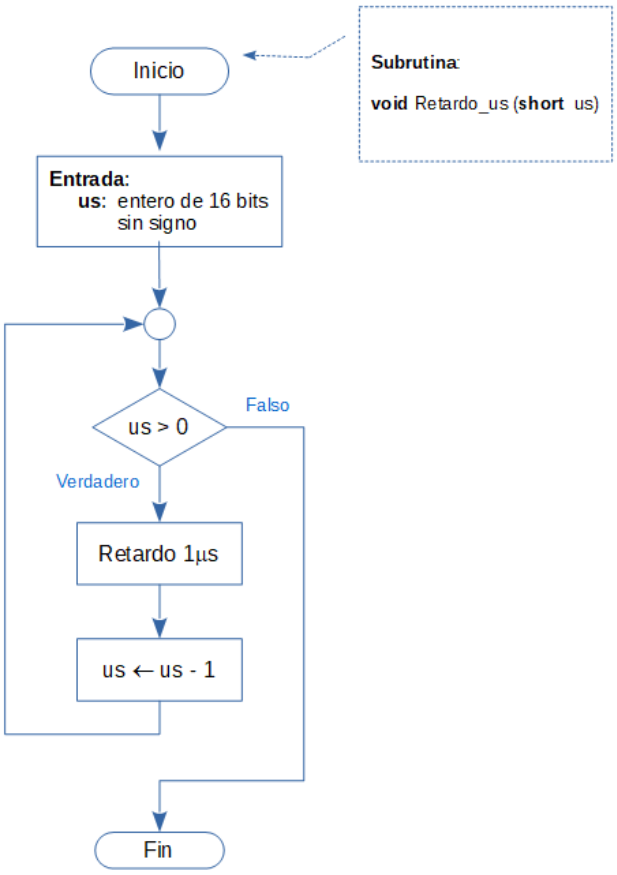
El parámetro de entrada `ms` es un entero 16-bits sin signo, y se almacenará el valor de su argumento en los registros R25:R24. Se valor representa la cantidad de milisegundo (ms) de retardo.

Análisis:

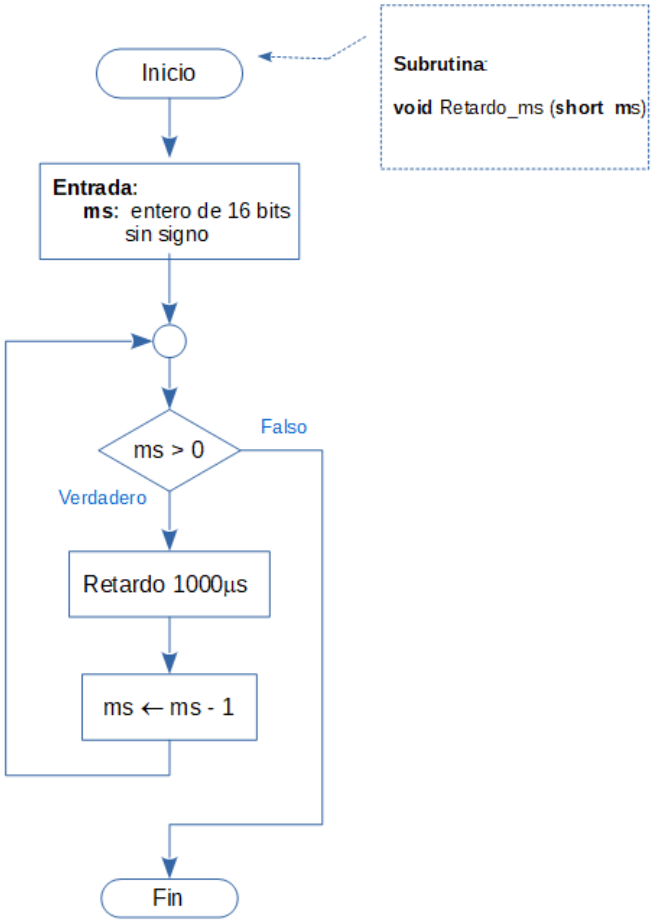
Como el MCU Atmega2560 tiene un reloj de 16MHz, además como la mayoría de las instrucciones se ejecutan en un ciclo de reloj, esto significa que cada instrucción tiene un tiempo de ejecución:

$$T_{inst}\frac{1}{16x10^6} = 0.0625\ \mu s$$

Es decir, se necesitan 16 instrucciones (con un tiempo de ejecución de un ciclo reloj) por cada microsegundo. El algoritmo para retardar dados `us` microsegundos se muestra en la siguiente figura:



El siguiente diagrama de flujo muestra el algoritmo para la rutina de retardo en milisegundos `Retardo_ms(ms);`



La codificación en lenguaje ensamblador, se debe hacer considerando las especificaciones que el compilador ha definido: los registros R18-R27 se pueden emplear libremente dentro de las rutinas, sin necesidad de guardar su estado (el código quién invoca a la rutina tiene la obligación de guardar su estado); el primer argumento de entrada de una subrutina se almacena en R24 si es de 8-bits y en R25:R24 si es de 16-bits. También es necesario especificar el identificador con el nombre de la subrutina como global, para que el ligador pueda visualizarlos.

El siguiente código muestra la codificación de la subrutina `Retardo_us(us)`.

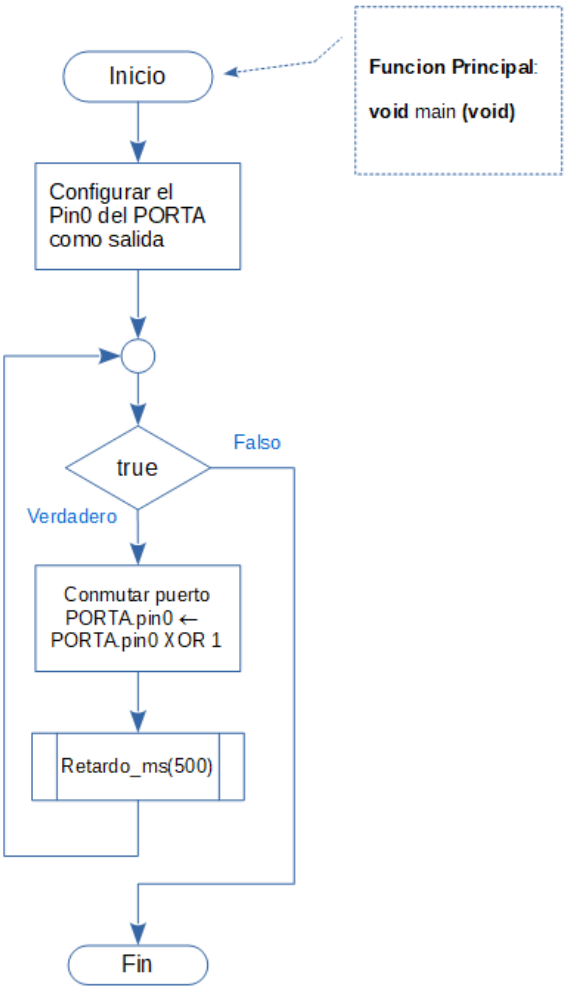
```
42  /* ------------------------------------------------------------------------------
43   * Subrutina:
44   *
45   * void Retardo_us(unsigned short us);
46   * Entrada: us: Cantidad de microsegundo de retardo
47   *              Entero de 16-bits sin signo
48   *              El argumento es almacenado en R25:R24
49   * ------------------------------------------------------------------------------
50   */
51
52          .global Retardo_us  ;define a la etiqueta Retardo_ms como global
53
54          .text           ;Inicia la sección de código
55
56  Retardo_us:
57
58  INI_WHILE_01:
59          MOVW    R20, R24            ; R21:R20 <-- R25:R24   (1 ciclo)
60          OR R21, R20                 ; R21 <-- R21 OR R20    (1 ciclo)
61          BREQ    FIN_WHILE_01        ; Mientras R21:R20 != 0 (1 ciclo)
62          NOP                         ; No operación          (1 ciclo)
63          NOP                         ; No hace nada          (1 ciclo)
64          NOP                         ; solo para retardo     (1 ciclo)
65          NOP                         ; No operación          (1 ciclo)
66          NOP                         ; No operación          (1 ciclo)
67          NOP                         ; No operación          (1 ciclo)
68          NOP                         ; No operación          (1 ciclo)
69          NOP                         ; No operación          (1 ciclo)
70          NOP                         ; No operación          (1 ciclo)
71          SBIW R24, 1                 ; R25:R24 <-- R25:R24 - 1   (2 ciclo)
72          RJMP INI_WHILE_01           ; Salta al inicio del while (2 ciclos)
73                                      ;--------------------------------------------
74                                      ; Total = (16 ciclos) x repeticion + 5 ciclos
75  FIN_WHILE_01:                       ;       = 1us x (R25:R24) + 0.3125 us
76          RET         ;Regreso de subrutina
77
```

En necesario hacer notar, que el ensamblador GNU GCC no acepta la escritura de pares de registros de 16-bits como R25:R24, a diferencia del ensamblador AVR que si lo hace. Por ejemplo, la instrucción `MOVW R21:R20, R24:R25` el ensamblador AVR (avrasm2) no marcará ningún error, a diferencia del ensamblador GNU GCC, que marcará un error, debido a que esta instrucción debe escribirse como: `MOVW R20, R25`

El siguiente código muestra la codificación de la subrutina `Retardo_ms(ms)`.

```
8   /* ------------------------------------------------------------------------------
9    * Subrutina:
10   *
11   * void Retardo_ms(unsigned short ms);
12   * Entrada: ms: Cantidad de milisegundo de retardo
13   *              Entero de 16-bits sin signo
14   *              El argumento es almacenado en R25:R24
15   * ------------------------------------------------------------------------------
16   */
17
18          .global Retardo_ms  ;define a la etiqueta Retardo_ms como global
19
20          .text           ;Inicia la sección de código
21
22  Retardo_ms:
23          MOVW    R26, R24            ; R27:R26 <-- R25:R24   (1 ciclo)
24  INI_WHILE_02:
25          MOVW    R20, R26            ; R21:R20 <-- R27:R26   (1 ciclo)
26          OR R21, R20                 ; R21 <-- R21 OR R20    (1 ciclo)
27          BREQ    FIN_WHILE_02        ; Mientras R21:R20 != 0 (1 ciclo)
28
29          LDI     R24, lo8(999)       ; R25:R24 <-- 999       (1 ciclo)
30          LDI     R25, hi8(999)       ; Parte alta            (1 ciclo)
31          CALL    Retardo_us          ; Llama a subrutina     (5 ciclos)
32                                      ; Retardo_us(1000)      (16x999 + 5 =
33                                      ;                        15,989 ciclos)
34          SBIW R26, 1                 ; R25:R24 <-- R25:R24 - 1   (2 ciclo)
35          RJMP INI_WHILE_02           ; Salta al inicio del while (2 ciclos)
36                                      ;-------------------------------------------
37                                      ; Total = (15,989 + 14 ciclos) x repeticion + 6 ciclos
38  FIN_WHILE_02:                       ;       = (16003)x repeticion + 0.375 us
39          RET ;Regreso de subrutina   ;       =  1.0001875ms x repeticion + 0.375 us
40
```

En la función principal `main()` se debe configurar y controlar el puerto A para que se encienda y apague el pin0 cada medio segundo, he de allí que su algoritmo es especificado como muestra el siguiente diagrama de flujo.



La codificación de la función principal se muestra en el siguiente código.

```
8   //Deficición de la dirección de los registros del puerto A
9   #define PORTA ((unsigned char *)0x0022)
10  #define DDRA ((unsigned char *)0x0021)
11  #define PINA ((unsigned char *)0x0020)
12
13  //Prototipo de subrutinas definidas como externas
14  extern void Retardo_ms(unsigned short ms);
15  extern void Retardo_us(unsigned short us);
16
17  int main(void)
18  {
19      //Configuración del pin0 del puerto A como salida
20      *DDRA = 0x01;
21      while (1)
22      {
23          // Conmuta el pin0, al escribir un 1 al bit0 del registro
24          // PINA.bit0 (el puerto está configurado como salida)
25          *PINA = 0x01;
26
27          Retardo_ms(500);
28      }
29  }
30
```

5. El display LCD 1602A

El módulo LCD 1602A (ver figura 5.1) es una pantalla para desplegar 16 caracteres alfanuméricos por 2 líneas, cada carácter esta representado por una matriz de 8x5 puntos, opcionalmente se puede configurar el despliegue de una sola línea con caracteres de 8x5 puntos o 11x5 puntos. Tiene una memoria RAM para despliegue (DDRAM) de 80 caracteres (40 caracteres por línea), una memoria ROM preprogramada con una fuente de 240 caracteres (*Character Generator ROM*, CGROM) y una memoria RAM para establecer hasta 8 caracteres personalizados (*Character Generator RAM, CGRAM*).

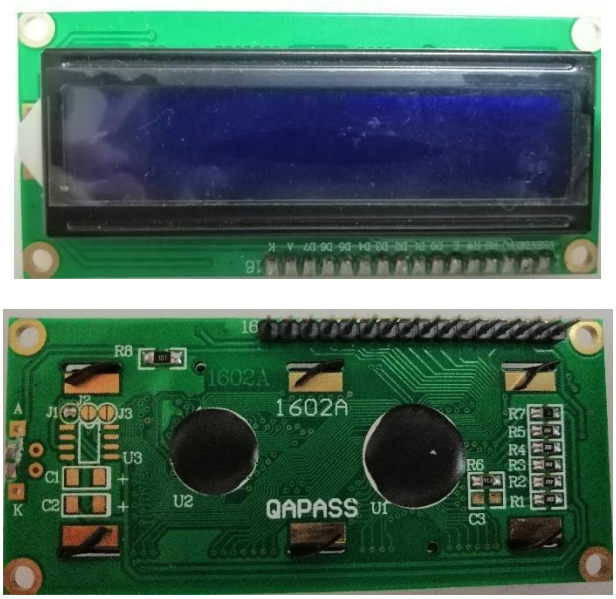


Figura 5.1. Display LCD 1602A alfanumérico de 16x2, vista frontal y posterior.

Incluye una interfaz de comunicación para microcontrolador (MCU) con un bus de 4-bits y 8-bits, la tabla 5.1 describe cada una de las señales de la interfaz de comunicación. Maneja valores de voltajes de alimentación de 5V e incluye luz LED de fondo con intensidad variable a partir de un potenciómetro 10kΩ (ver figura 5.2).

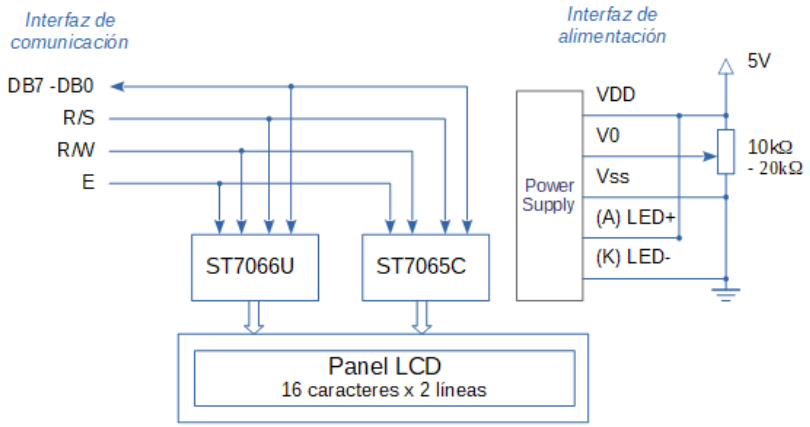


Figura 5.2. Diagrama de bloques del display LCD 1603A

Tabla 5.1. Señales de la interface del display LCD 1602A

Nombre de la señal	Tipo de señal Entrada/Salida	Descripción
E (Enable)	Entrada	Establece el inicio de una lectura o escritura
R/W (Read/Write)	Entrada	Selecciona el modo de operación del bus Escritura/Lectura 0: Escritura 1: Lectura
RS (Register Select)	Entrada	Selecciona el registro a escribir o leer. En combinación con R/W puede seleccionar el registro de Instrucción (IR), el registro de datos (DR) o el registro contador de direcciones (AC, Address Counter). 0: Selecciona IR (modo escritura) o AC (modo lectura) 1: Selecciona el registro DR (Lectura o Escritura)
Db7 – Db0	Entrada y Salida	Bus de datos bidireccional con capacidad de tercer estado (alta impedancia). Se usan para la transferencia y recepción de información entre el MCU y el módulo LCD 1602A. Cuando el módulo está configurado con interfaz de 4 bits los bits Db3 al Db0 no se emplean.

La figura 5.3 muestra la interfaz de comunicación entre un microcontrolador y el display LCD 1602A y las figuras 5.4 y 5.5 representan las señales de control y sincronización para la escritura y lectura de información entre el MCU y el LCD.

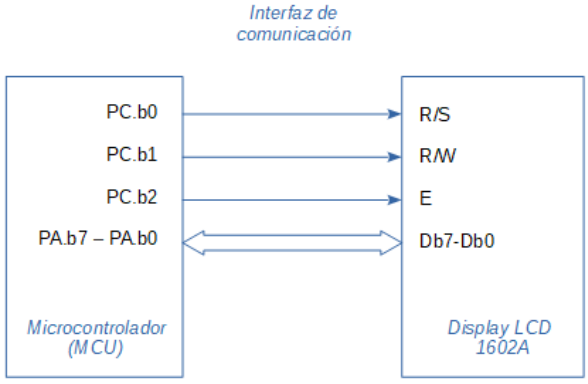


Figura 5.3. Interfaz de comunicación entre el MCU y el LCD 1602A

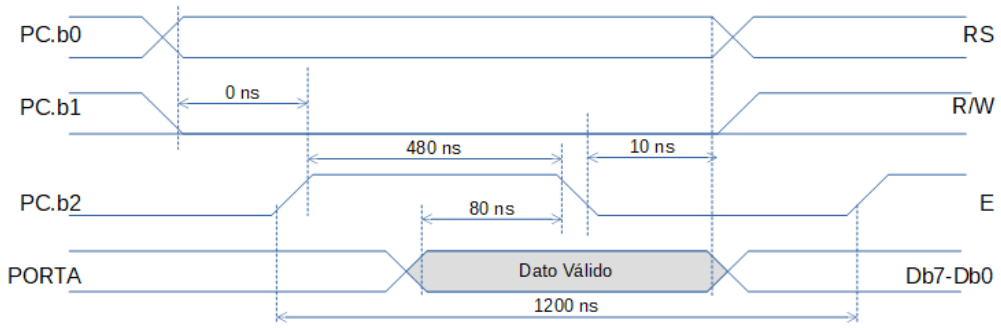


Figura 5.4 Señales de escritura de un dato del MCU al LDC 1602A

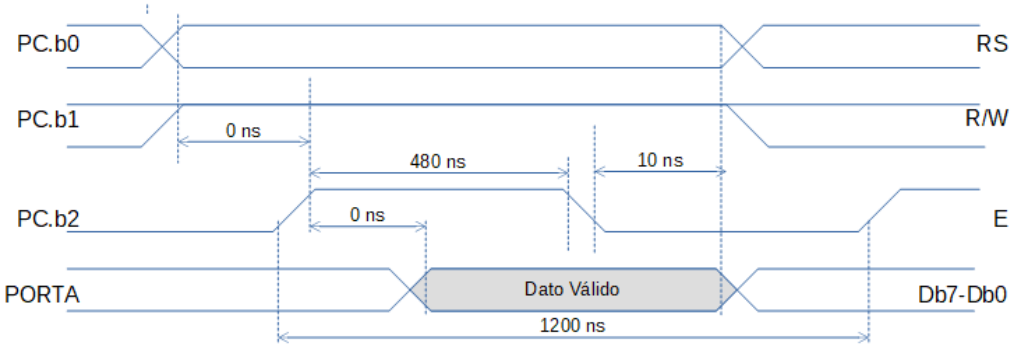


Figura 5.5 Señales de lectura desde el LDC 1602A hacia el MCU

5.1. Descripción de los registros del LCD 1602A

Para la operación del LCD, el driver contiene tres registros que le ayudan a controlar el comportamiento del LDC: Registro de Instrucción (IR), recibe el código de una instrucción enviada del MCU al LCD; el registro de datos (DR), almacena temporalmente el datos que se escribe o lee de las memorias DDRAM (*Data Display RAM*) o CGRAM (*Character Generator RAM*); Registro contador de direcciones (AC), almacena la dirección de las memorias DDRAM/CGRAM, el registro se incrementa automáticamente en 1, cuando el MCU lee este registro, el bus de datos envía Db6 al Db0 este registro y en el Db7 se envía la bandera de ocupado (BF, *Busy Flag*).

El display solo puede desplegar 16 caracteres por línea a la vez, es configurable un desplazamiento automático a la izquierda o derecha para mostrar los restantes caracteres almacenador en la DDRAM.

Si el display está configurado en una sola línea, los caracteres se almacenan en forma lineal en la memoria DDRAM, desde la dirección 0x00 hasta la dirección 0x4F dando un total de 80 caracteres (ver figura 5.6).

Por otro lado, si se configura el display en modo de dos líneas, la dirección de los caracteres cambia, los primeros 40 caracteres (correspondientes a la primera línea), se almacenarán en las direcciones de memoria DDRAM 0x00 hasta la localidad 0x27 y la segunda línea se almacenarán en las direcciones 0x40 hasta 0x67 (ver figura 5.7).

Display Position (Digit)	1	2	3	4	5	6	.....	78	79	80
DDRAM Address	00	01	02	03	04	05	.....	4D	4E	4F

Figura 5.6. Direcciones de memoria de despliegue, configuración de una línea.

Display Position	1	2	3	4	5	6	.....	38	39	40
DDRAM Address (hexadecimal)	00	01	02	03	04	05	.....	25	26	27
	40	41	42	43	44	45	.....	65	66	67

Figura 5.7. Direcciones de memoria de despliegue, configuración de dos líneas.

La figura 5.8, muestra algunos de los caracteres almacenados en la memoria CGROM

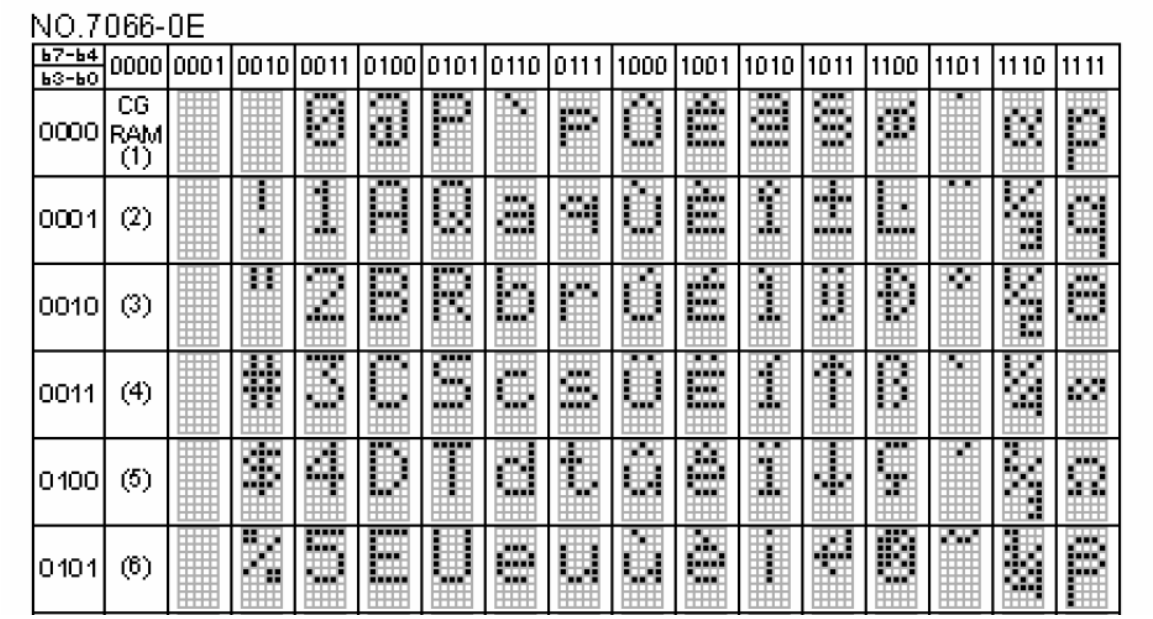


Figura 5.8, alguno de los caracteres grabados en la memoria CGROM del display

La memoria CGRAM (*carácter Generator RAM*) permite establecer hasta 8 caracteres de 8x5 puntos personalizados. Estos caracteres son mapeados en la tabla de caracteres CGROM en las direcciones 0 hasta 7 y se repite en las direcciones 8 hasta 15. La tabla 5.1 muestra la relación entre el código DDRAM, las direcciones de memoria CGRAM y los datos binarios que definen los caracteres.

Character Code (DDRAM Data)								CGRAM Address						Character Patterns (CGRAM Data)							
b7	b6	b5	b4	b3	b2	b1	b0	b5	b4	b3	b2	b1	b0	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	-	0	0	0	0	0	0	0	0	0	-	-	-	1	1	1	1	1
					0	0	0				0	0	1				0	0	1	0	0
					0	0	0				0	1	0				0	0	1	0	0
					0	0	0				0	1	1				0	0	1	0	0
					0	0	0				1	0	0				0	0	1	0	0
					0	0	0				1	0	1				0	0	1	0	0
					0	0	0				1	1	0				0	0	1	0	0
					0	0	0				1	1	1				0	0	0	0	0
0	0	0	0	-	0	0	1	0	0	1	0	0	0	-	-	-	1	1	1	1	0
					0	0	1				0	0	1				1	0	0	0	1
					0	0	1				0	1	0				1	0	0	0	1
					0	0	1				0	1	1				1	1	1	1	0
					0	0	1				1	0	0				1	0	1	0	0
					0	0	1				1	0	1				1	0	0	1	0
					0	0	1				1	1	0				1	0	0	0	1
					0	0	1				1	1	1				0	0	0	0	0

Tabla 5.2 Relación de entre las direcciones CGRAM, código asociado al carácter DDRAM y el patrón de bits que definen el carácter.

5.2. Descripción de las instrucciones soportadas por el LCD 1602A

En el siguiente apartado se describe las instrucciones que soporta el display LCD 1602A

- **Limpiar display**. Limpia el display escribiendo el carácter 0x20 (carácter de espacio) en todas las direcciones de memoria DDRAM e inicializa el registro contador de direcciones (AC) a la dirección 0x00. Regresa el cursor al origen, establece el modo de incremento del contador a 1 (I/D = 1).

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	0	1

- **Regresa a inicio**. Regresa el cursor al inicio y establece el contador de direccionas DDRAM a 0x00, el contenido de la memoria no cambia.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	0	1	x

- **Establece el modo de entrada**. Establece la dirección de movimiento del cursor y el display.
  I/D = 1: El cursor se mueve hacia la derecha, parpadea, la dirección de la DDRAM se incrementa en 1
  I/D = 0: El cursor se mueve hacia la izquierda, parpadea, la dirección de la DDRAM se decrementa en 1.
  S = 1: Se activa el corrimiento del display de acuerdo con I/D.
  S = 0: Se desactiva el corrimiento del display.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	0	1	I/D	S

- **Display ON/OFF**. Controla el display, Cursor y su parpadeo

  D = 1: Se enciende el display.
  D = 0: Se apaga el display.
  C = 1: Se activa el cursor.
  C = 0: Se desactiva el cursor.
  B= 1: Se activa el parpadeo del cursor.
  B = 0: Se desactiva el parpadeo del cursor.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	0	1	D	C	B

- **Corrimiento del display o el cursor**. Corrimiento del cursor o el display hacia la izquierda o derecha. En el modo de operación de 2 líneas, el cursor se mueve a la segunda línea después del 40 digito.

S/C	R/L	Descripción	Valor del registro AC
0	0	Corrimiento del cursor a la izquierda	ACC = AC - 1
0	1	Corrimiento del cursor a la derecha	ACC = AC + 1
1	0	Corrimiento del display a la izquierda	ACC = AC
1	1	Corrimiento del display a la derecha	ACC = AC

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	0	1	S/C	R/L	x	x

- **Configuración el modo de operación.** Establece el modo de operación 1 ó 2 líneas, tamaño de fuente, el modo de operar el bus de datos 8-bits ó 4-bits.

  DL = 1: Establece el bus de datos de 8-bits.
  DL = 0: Establece el bus de datos de 4-bits.
  N = 1: Modo de operación de 2 líneas del display.
  N = 0: Modo de operación de 1 líneas del display.
  F = 1: Representación del carácter de 5x11 puntos. Esta función sólo es válida para N=0
  F = 0: Representación del carácter de 5x8 puntos.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	0	1	DL	N	F	x	x

- **Establecer la dirección de CGRAM.** Activa la memoria CGRAM para escritura o lectura. Establece la dirección de la memoria CGRAM al registro AC (*Address Register*).

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0

- **Establecer la dirección de DDRAM.** Activa la memoria DDRAM para escritura o lectura. Establece la dirección de la memoria DDRAM al registro AC (*Address Register*). Para el modo de despliegue de 1 línea, la dirección DDRAM es de 0x00 a 0x47, para el modo de 2-líneas la dirección es 0x00 a 0x27 (primera línea) y 0x40 a 0x67 (segunda línea).

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

- **Lectura del registro AC y bandera de ocupado.** Lectura del registro contador de direcciones (AC) que almacena las direcciones para las memorias DDRAM y CGRAM.

  BF = 1: Ocupado, el dispositivo está realizando una operación interna (no puede aceptar una instrucción).
  BF = 0: No ocupado, el dispositivo puede aceptar instrucciones.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0

- **Escritura de un dato hacia DDRAM o CGRAM.** Escribe un dato de 8-bits a la memoria seleccionada DDRAM o CGRAM. Antes de la escritura, se debe seleccionar el tipo de memoria con las instrucciones para establecer la dirección del registro AC. Después de una operación de escritura el registro AC es incrementado a 1 de forma automática.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	1	0	D7	D6	D5	D4	D3	D2	D1	D0

- **Lectura de un dato desde la DDRAM o CGRAM.** Lee un dato de 8-bits desde la memoria seleccionada DDRAM o CGRAM. Antes de la lectura, se debe seleccionar el tipo de memoria con las instrucciones para establecer la dirección del registro AC. Después de una operación de lectura el registro AC es incrementado a 1 de forma automática.

	RS	RW	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Code	1	1	D7	D6	D5	D4	D3	D2	D1	D0

5.3. Proceso de inicialización del LCD 1602A

Cuando se enciende el dispositivo LCD 1602A, automáticamente se genera un evento de restablecimiento (reset), la bandera de ocupado (BF) se mantiene a 1 mientras no termine este proceso, ya terminado cambia la bandera BF a 0. El estado del dispositivo queda configurado como se enlista a continuación.

- Display limpiado
- Modo de operación: **DL = 1**, interface de datos de 8-bits; **N=0**, despliegue de 1 línea; **F=0**, tamaño de 8x5 puntos por cada carácter.
- Control de encendido / Apagado: **D = 0**, display apagado; **C = 0**, cursor apagado; **B = 0**, parpadeo apagado.
- Modo de entrada: **I/D = 1**, Incremento del contador AC a 1; **S = 0**, sin corrimiento.

El siguiente algoritmo muestra el procedimiento de inicialización y configuración del display LCD 1602A.

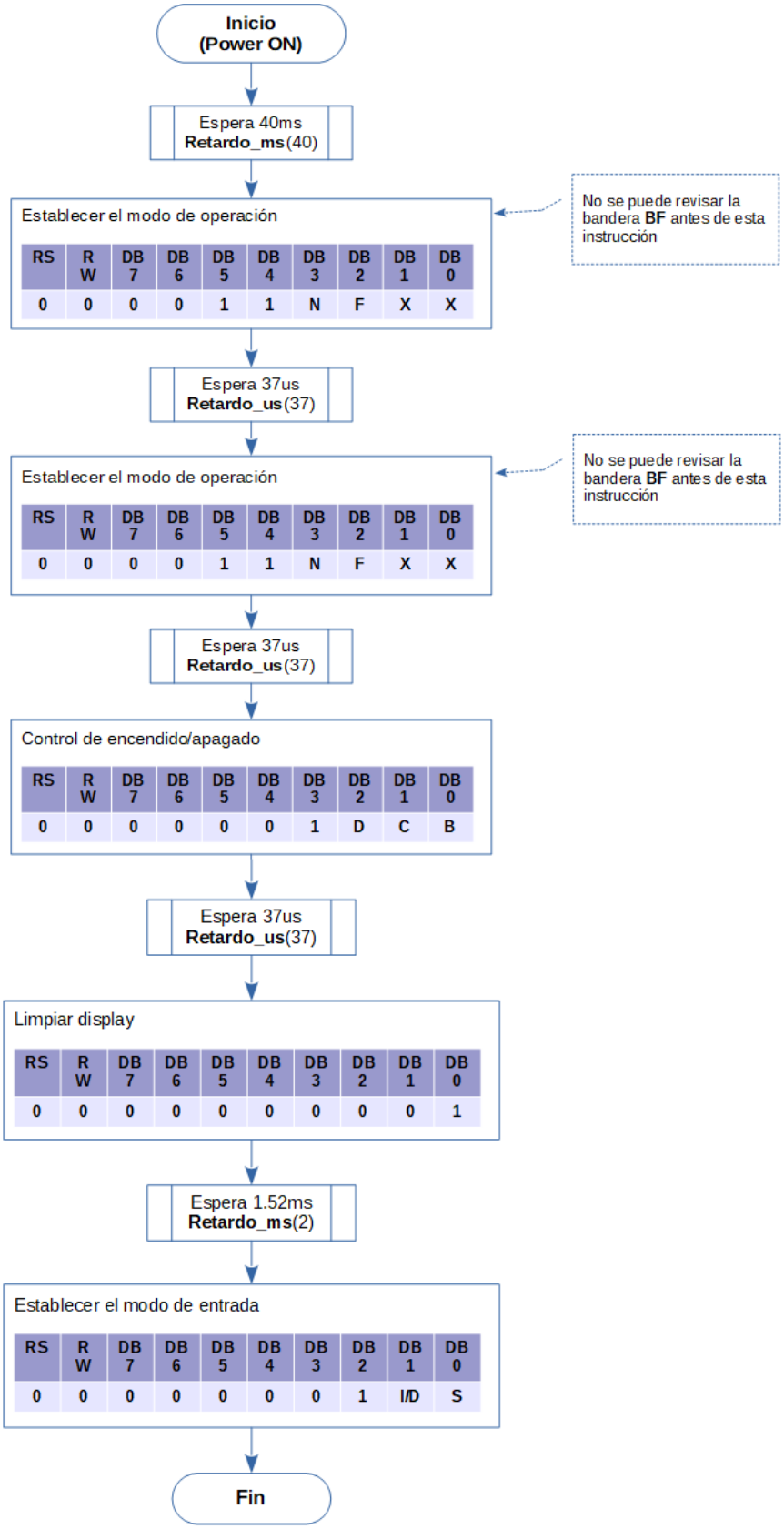
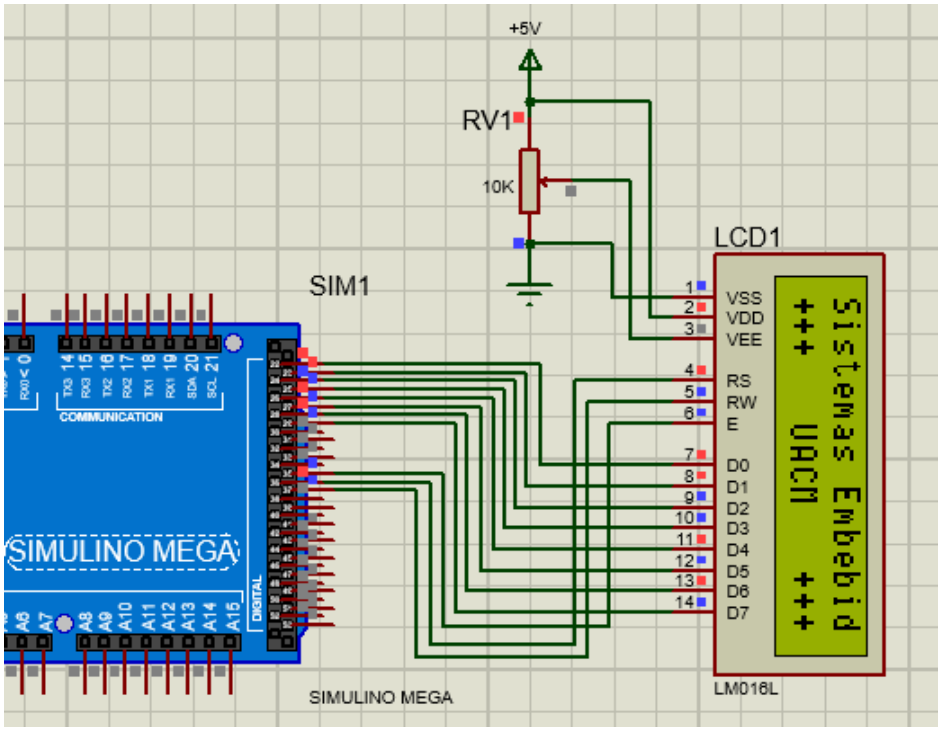


Figura 5.9, Algoritmo de inicialización del display LCD 1602A

6. Desarrollo de la práctica

6.1. Conectar la tarjeta Arduino Mega con el display LCD1602A (en Proteus LDC LM016L) utilizando el puerto A como bus de datos bidireccional y el puerto C como bus de control. La siguiente tabla y figura muestran las conexiones realizadas.

Atemga 2560 Puertos	Interface LCD 1602A
PORTA.b7 al PORTA.b0	Db7 al Db0
PORTC.b0	R/W
PORTC.b1	RS
PORTC.b2	E



6.2. Elaborar un conjunto de rutinas en lenguaje ensamblador que sirvan como librería para controlar el desplegado de texto en el display LCD 1602.

La librería deberá tener las siguientes subrutinas.

- **LCD_WriteComando**(unsigned short comando)

Esta rutina envía un comando al display. Genera las señales para la transmisión de un dato mediante los puertos A y C. La variable comando contiene la instrucción a enviar (b9,b8,b7,b6,b5,b4,b3,b2,b1,b0) → (RS, R/W, DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)

PORTA(bit7-bit0) → LCD(DB7-DB0)

PORTC(bit2-bit0) → LCD(E, RS, R/W)

- unsigned char byte **LD_Read_BFAddress**()

Rutina para leer la bandera de ocupado (BF) bit7 y el estado del registro contador de direcciones AC bit6 hasta el bit0.

- **LCD_Inicializar**()

Inicializar y configurar el dispositivo LCD para que utilice una interface de datos de 8-bits, configura en modo de 2 líneas de texto y tamaño de carácter de 8x5 puntos.

- **LCD_Clear**()

Limpia la pantalla LCD.

- **LCD_CursorToHome**()

Sitúa el cursor al inicio.

- **LCD_SetCursor**(unsigned char reng, unsigned char col)

Coloca el cursor en el renglón y columna especificado. El parámetro de entrada `reng` establece el número de la línea: 0 para la primera línea y 1 para la segunda línea. El parámetro `col` establece la posición en la línea, los valores aceptados son 0 al 39.

- **LCD_WriteChar**(char caracter)

Escribe un carácter en la posición actual del cursor

- **LCD_WriteString**(char *pCadena)

Escribe una cadena de caracteres al display, la cadena debe terminar con el carácter nulo '\0'. El parámetro pCadena guarda la dirección en memoria de programa donde se guardará la cadena a desplegar.

6.3. Codificar el siguiente conjunto de rutinas en lenguaje ensamblador para habilitar o deshabilitar ciertas funcionalidades del display.

- **LCD_HabilitarDisplay**()/**LCD_DeshabilitarDisplay**()

Habilitar/Deshabilitan el display.

- **LCD_HabilitarCursor**()/**LCD_DeshabilitarCursor**()

Habilitar/Deshabilitan el cursor .

- **LCD_HabilitarParpadeo**()/**LCD_DeshabilitarParpadeo**()

Habilitar/Deshabilitar el parpadeo del cursor.

- **LCD_HabilitarAutoScroll**()/**LCD_DeshabilitarAutoScroll**()

Habilitar/Deshabilitar el desplazamiento automático.

- **`LCD_ScrollDisplayIzq`**`()`/**`LCD_ScrollDisplayDerecha`**`()`

Establece la dirección del desplazamiento del display izquierda o derecha.

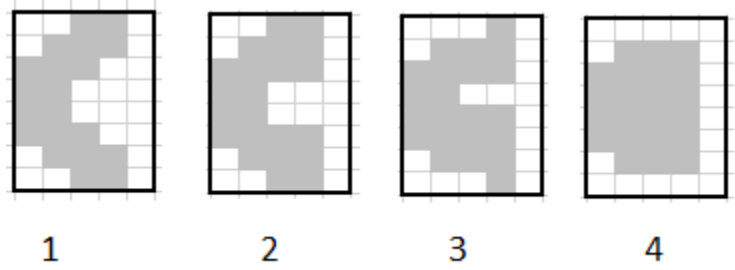
6.4. Elaborar un programa en lenguaje C que muestre el texto “Sistemas embebidos” en la primera línea y en la segunda línea el texto “UACM”, usando las subrutinas programadas.

6.5. Codificar la rutina para establecer uno de los 8 caracteres personalizados.

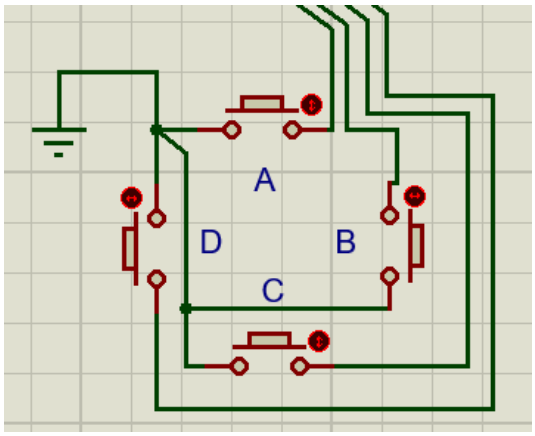
- **`LCD_SetChar`**`(char dir, char *pPatron)`

Establece un carácter personalizado en la memoria CGRAM. El parámetro de entrada `dir`, determina la dirección de la memoria CGRAM y sólo puede ser un valor entre 0 y 7; el parámetro `pPatron` determina la dirección de memoria de programa donde se ha guardado un arreglo de 8 bytes que representan el carácter personalizado (ver tabla 5.2).

6.6. Con la rutina del inciso previo, crear los siguientes caracteres personalizados



6.7. Elaborar un programa en lenguaje C que realice el juego de PAC-MAN donde se podrá comer las letras escritas en el display LCD, PAC-MAN comenzará a comer en la posición 0,0 y dependiendo del botón pulsado, se moverá a la izquierda, derecha, arriba y abajo; siempre respetando los límites del display de 2x16. Los botones *push-button* para controlar a PAC-MAN, se pueden conectar en un puerto de entrada diferente a los puertos ocupados por el display LCD. El siguiente circuito es el que se puede utilizar como interfaz de entrada. El algoritmo primero mostrará un texto en el display, después entrará en un ciclo donde revisará cada 300ms si hay algún botón pulsado, si lo hay, entonces se hace el movimiento de PAC-MAN en la posición indicada por el botón y con la siguiente imagen de la animación. Cuando PAC-MAN pasa en la posición de una letra, se dibuja en su posición y cuando sale, el carácter se transforma en espacio. después se tiene que esperar 300ms y se repite el ciclo hasta completar el juego de comerse todas las letras.



Bibliografía

- *Atmel 2549 8-bit AVR Microcontroller ATmega640/1280/1281/2560/2561 datasheet*
- *Atmel 0856 AVR Instruction Set Manual*
- *Atmel 42167 Atmel Studio User Guide*
- *Arduino mega2560 R3 diagrama esquemático.*
- *Using the GNU Compiller Collection, https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc.pdf*
- *Atmel AT1886: Mixing Assembly and C with AVRGCC*
- *Mixing C and Assembly languaje programs, William Barnekow, 2007*
- *Dot Matrix LCD Controller/Controller ST7066U*

Código Fuente

```
2   /*
3    * LCD_1602A_Biblioteca.s
4    *  Author: Omar Nieto Crisóstomo
5    */
6
7               .EQU PINA,0x00
8               .EQU DDRA,0x01
9               .EQU PORTA,0x02
10
11              .EQU PINB,0x03
12              .EQU DDRB,0x04
13              .EQU PORTB,0x05
14
15              .EQU PINC,0x06
16              .EQU DDRC,0x07
17              .EQU PORTC,0x08
18
```

```
19  /* ------------------------------------------------------------------------
20   * void LCD_WriteComando(unsigned short comando);
21   * Entrada:
22   *     comando, comando enviado al LCD compuesto de 10 bits
23   *     (RS,RW,DB7,DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0) en los registros R25:R24
24   * Esta rutina envía un comando al display LCD.
25   * Genera las señales para la transmisión de una instrucción por medio de los
26   * Puerto A y C.
27   * En el puerto A tiene conectado el bus de datos PORTA(bit7-bit0) => LCD(DB7-DB0)
28   * El puerto C se conectan las seguientes señales PORTC(bit2-bit0) => LCD(E, RS, R/W)
29   * ------------------------------------------------------------------------
30   */
31              .global LCD_WriteComando
32
33              .text
34  LCD_WriteComando:
35              // Se configuran los puertos A y C como salida
36              LDI R20, 0x07
37              OUT DDRC, R20
38              CLR R20
39              OUT PORTC, R20
40              LDI R20, 0xFF
41              OUT DDRA, R20
42
43              //R24 => PORTA(bit7-bit0) => LCD(DB7-DB0)
44              //R25 => PORTC(bit2-bit0) => LCD(E, RS, R/W)
45
46              //Se envia las señales hacia el LCD
47              CBR R25, 0xFC           ; RS: X=======================
48              OUT PORTC, R25          ; RW: X=======X___________________x========
49              SBI PORTC, 2            ;  E: X_______x----480ns-------
50              OUT PORTA, R24          ; DB: X__________x=========DATOS====
51              //Cada instrucción tarda 62.5ns entonces
52              //se necesitan 480/62.5 => 8 instrucciones para desactiva la señal E
53              NOP
54              NOP
55              NOP
56              NOP
57              NOP
58              NOP
59              NOP
60              NOP
61              CBI PORTC, 2            ;  E: X_______x----480ns-------x___
62              //se necesitan 1200/62.5 => 20 instrucciones
63              //para enviar un nuevo comando, ya se han ejecutado 10, faltan 10 mas
64              NOP
65              NOP
66              NOP
67              NOP
68              NOP
69              NOP
70              NOP
71              NOP
72              NOP
73              RET
```

```
/* ------------------------------------------------------------------------
 * unsigned char LCD_Read_BFAddress();
 * Salida:
 *     Estado del registro contador de direcciones (AC) y la bandera de ocupado (BF)
 *     (BF,ACB6,DB5,DB4,DB3,DB2,DB1,DB0)
 * ------------------------------------------------------------------------
 */
            .global LCD_Read_BFAddress

            .text
LCD_Read_BFAddress:
            // Se configuran el puerto A como entrada y C como salida
            LDI R20, 0x07
            OUT DDRC, R20
            CLR R20
            OUT PORTC, R20
            OUT DDRA, R20

            //R24 => PORTA(bit7-bit0) => LCD(DB7-DB0)
            //R25 => PORTC(bit2-bit0) => LCD(E, RS, R/W)

            //Se envia las señales hacia el LCD
            LDI R25, 0x01           ; RS: X_____________________________________
            OUT PORTC, R25          ; RW: X_X--------------------------x_____
            SBI PORTC, 2            ;  E: X_______x----480ns-------x
                                    ; DB: X__________x=========DATOS====
            //Cada instrucción tarda 62.5ns entonces
            //se necesitan 480/62.5 => 8 instrucciones para desactiva la señal E
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            IN R24, PINA
            NOP
            CBI PORTC, 2            ;  E: X_______x----480ns-------x___
            //se necesitan 1200/62.5 => 20 instrucciones
            //para enviar un nuevo comando, ya se han ejecutado 10, faltan 10 mas
            CLR R20
            OUT PORTC, R20
            NOP
            NOP
            NOP
            NOP
            NOP
            NOP
            RET
```

```
/* ------------------------------------------------------------------------------
 * void LCD_Inicializar();
 *     Inicializa el LCD con los siguientes parámetros
 *     bus de datos 8-bits, modo de 2 linea y 8x5 puntos por caracter
 *     cursor se deplaza izquierda y en parpadeo
 * ------------------------------------------------------------------------------
 */
            .global LCD_Inicializar

            .text
LCD_Inicializar:
            // Retardo de 40ms
            CLR R25
            LDI R24, 40
            CALL Retardo_ms

            // Establece el modo de operación
            // (RS,RW,  DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)
            // ( 0, 0,   0, 0,  1,  DL,  N   F,  0   0)
            //          DL = 1: 8bits  Bus de datos
            //          N  = 1: modo de 2 lineas
            //          F  = 0: modo de fuente de 8x5 puntos
            CLR R25
            LDI R24, 0x38
            CALL LCD_WriteComando
            // Retardo de 37us
            CLR R25
            LDI R24, 37
            CALL Retardo_us
            // Establece el modo de operación
            // ( 0, 0,   0, 0,  1,  DL,  N   F,  0   0)
            //          DL = 1: 8bits  Bus de datos
            //           N = 1: modo de 2 lineas
            //           F = 0: modo de fuente de 8x5 puntos
            CLR R25
            LDI R24, 0x38
            CALL LCD_WriteComando
            // Retardo de 37us
            CLR R25
            LDI R24, 37
            CALL Retardo_us
            // Control de encendido y apagado
            // (RS,RW,  DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)
            // ( 0, 0,   0,  0,  0,  0,  1,  D,  C,  B)
            //           D = 1: Display ON
            //           C = 1: Cursor ON
            //           B = 1: Parpadeo ON
            CLR R25
            LDI R24, 0x0F
            CALL LCD_WriteComando
            // Retardo de 37us
            CLR R25
            LDI R24, 37
            CALL Retardo_us
            // Limpiar display
            // (RS,RW,  DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)
            // ( 0, 0,   0,  0,  0,  0,  0,  0,  0,  1)
            CLR R25
            LDI R24, 0x01
            CALL LCD_WriteComando
            // Retardo de 1.52ms
            LDI R25, hi8(1520)
            LDI R24, lo8(1520)
            CALL Retardo_us
            // Establece el modo de entrada
            // (RS,RW,  DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)
            // ( 0, 0,  0,  0,  0,  0,  0,  1,  I/D,  S)
            //           I/D = 1: Cursor se desplaza a la derecha
            //             S = 1: Display se desplaza acorde a I/D
            CLR R25
            LDI R24, 0x0F
            CALL LCD_WriteComando

            RET
```

```
127  /* ------------------------------------------------------------------------------
128   * unsigned char byte LCD_EstaOcupado();
129   * Revisa si el LCD se encuentra ocupado, es decir una operación está en curos
130   * Salida:
131   *     R24: Bandera de ocupado. 1-ocupado, 0-No ocupado
132   * ------------------------------------------------------------------------------
133   */
134              .global LCD_EstaOcupado
135
136              .text
137  LCD_EstaOcupado:
138              CALL LCD_Read_BFAddress
139              BST R24, 7          ;Guarda el bit R24.b7 en la bandera T
140              CLR R24
141              BLD R24, 0          ;Lee la bandera T y la guarda en el bit R24.b0
142              RET
```

```
144  /* ------------------------------------------------------------------------------
145   * void LCD_EsperarHastaNoOcupado();
146   * Revisa el estado del LCD y espera hasta que no esté ocupado
147   * ------------------------------------------------------------------------------
148   */
149              .global LCD_EsperarHastaNoOcupado
150
151              .text
152  LCD_EsperarHastaNoOcupado:
153              PUSH R24
154              PUSH R25
155  LCD_Esperar_eti1:
156              CALL LCD_Read_BFAddress
157              BST R24, 7             ;Guarda el bit R24.b7 en la bandera T
158              BRTS LCD_Esperar_eti1  ;Salta si la bandera T está activado
159              POP R25
160              POP R24
161              RET
```

```
/* ------------------------------------------------------------------------
 * void LCD_SetCursor(unsigned char reng, unsigned char col);
 * Sitúa el cursor en el rengón y columna dada
 * Entrada:
 *     reng, (R24) Número del renglón 0 para la primera línea y 1 para la segunda linea
 *      col, (R22) Número de columna, valores posibles 0 hasta 39
 * ------------------------------------------------------------------------
 */
            .global LCD_SetCursor

            .text
LCD_SetCursor:
            PUSH R24        ;Guarda los registros R24 y R22 en la pila
            PUSH R22
            CALL LCD_EsperarHastaNoOcupado   ;Espera hasta que el LCD pueda recibir comandos
            POP R22         ;Restaura los registros R24 y R22 de la pila
            POP R24

            // reg -->R24, col --> R22
            // Para dos lineas
            // Si reng == 0
            //     AC <-- col
            // Si reng == 1
            //     AC <-- 0x40 + col
            // Establece la dirección de la memoria DDRAM
            // (RS,RW,  DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)
            // ( 0, 0,    1,AC6,AC5,AC4,AC3,AC2,AC1,AC0)
            CPI R24, 0
            BREQ LCD_SET_CUR1           ; if (R24 != 0)
            LDI R20, 0x40           ;       R22 <-- R22 + 0x40
            ADD R22,R20
LCD_SET_CUR1:
            SBR R22, 0x80           ;R22.b7 <-- 1
            CLR R25
            MOV R24,R22
            CALL LCD_WriteComando
            RET
```

```
/* ------------------------------------------------------------------------
 * void LCD_WriteChar(unsigned char car);
 * Escribe un caracter en el display, en la posición actual del cursor
 * Entrada:
 *     car, (R24) Caracter a escribir en la memoria DDRAM de acuerdo a la tabla CGROM
 * ------------------------------------------------------------------------
 */
            .global LCD_WriteChar

            .text
LCD_WriteChar:
            PUSH R24        ;Guarda el registro R24 a la pila
            CALL LCD_EsperarHastaNoOcupado   ;Espera hasta que el LCD pueda recibir comandos
            POP R24         ;Restaura el registro R24 de la pila
            // car -->R24
            // Escribe un caracter en la memoria DDRAM
            // (RS,RW,  DB7,DB6,DB5,DB4,DB3,DB2,DB1,DB0)
            // ( 1, 0,   D7, D6, D5, D4, D3, D2, D1, D0)
            LDI R25, 0x02
            CALL LCD_WriteComando
            RET
```

```
299  /* ---------------------------------------------------------------------------
300   * void LCD_WriteString(const __flash unsigned char *pCad, unsigned char N);
301   * Escribe una cadena de caracteres de tamaño N en el display, a partir de la posición
302   * actual del cursor.
303   * Entrada:
304   *     pCar, (R25:R24) Apuntador a memoria de programa donde se encuentra
305   *           la cadena de caracteres
306   *        N, (R23:R22) Número de caracteres en la cadena
307   * ---------------------------------------------------------------------------
308   */
309              .global LCD_WriteString
310
311              .text
312  LCD_WriteString:
313              // pCar -->R25:R24, N --> R22
314
315              MOVW R30, R24       ; Z <-- R25:R24
316  LCD_write_string_et1:
317              CPI R22, 0
318              BREQ LCD_write_string_et2
319              PUSH R22
320              LPM R24, Z+
321              CALL LCD_WriteChar
322              POP R22
323              SUBI R22, 1
324              RJMP LCD_write_string_et1
325
326  LCD_write_string_et2:
327              RET
```