Especificación del lenguaje ccz80

versión 3.1.3

Compilador cruzado para Z80

http://ccz80.webcindario.com

Introducción

Se trata de un lenguaje basado en lenguaje C para crear código ensamblador o código binario, a partir de un código fuente en lenguaje ccz80, para ordenadores basados en un Zilog Z80 (Amstrad CPC/PCW, Spectrum 16/48/+2/+3, sistemas MSX, ...). El compilador cruzado ccz80 permite programar desde el entorno Windows, Linux o MacOS, aprovechando la comodidad y ventajas que tienen éstos, generando código ensamblador o binario para ordenadores basados en Z80.

El compilador está desarrollado en lenguaje C#, por lo que es necesario instalar Microsoft .NET Framework 2.0 o mayor en el ordenador para utilizar ccz80 o bien usar las utilidades WINE o MONO en Linux o MONO en MacOS.

Para utilizar este compilador es necesario realizar los siguientes pasos en el ordenador (PC, Mac, etc.):

1. Escribir un programa en lenguaje ccz80 y salvarlo en un fichero de texto (aconsejable con extensión .ccz80).

2. Compilar el programa desde la línea de comandos utilizando la sintaxis:

   ```
   ccz80 <fichero fuente> [/org=<dirección>] [/asm]
         [/include=<lista-rutas>] [/post=<aplicación> …]
   ```

   donde <fichero fuente> es el programa fuente escrito y salvado en un primer momento, opcionalmente <dirección> es la dirección de memoria de inicio para el código objeto creado donde se ejecutará el programa (esta dirección es 0 si no se especifica). Si la opción /asm se indica el compilador sólo genera el código ensamblador equivalente al código fuente ccz80, y no se genera el fichero binario, qué sí se genera si no se indica

esta opción. Con la opción /include se indican las rutas (separadas por punto y coma) donde se buscarán los ficheros indicados en la sentencias include y datafile. Se pueden especificar varias opciones /post para lanzar aplicaciones, con sus parámetros, tras terminar la compilación, si ha sido correcta; se irán lanzando en el mismo orden que aparezcan en la línea de órdenes y sólo se lanzará una aplicación si la anterior ha producido un código de salida 0 (se recomienda encerrar entre comillas dobles cada opción /post=<aplicación> si la aplicación incluye espacios para separar parámetros).

3. Cargar y ejecutar el fichero binario creado con el mismo nombre que el código fuente y extensión .bin en un emulador o un ordenador real basado en Z80.

El compilador ccz80 devuelve un valor para ERRORLEVEL 0 si la compilación es correcta, y 1 si no lo es.

Descripción del lenguaje ccz80

Tipos de datos

- **byte**: número en el rango 0 a 255
- **word**: número en el rango 0 a 65535

Se puede usar un valor byte o word como un puntero utilizando los operadores * o **.

La conversión entre ambos tipos es automática en cualquier expresión.

Los números flotantes no existen. Es necesario crear un conjunto de funciones para convertir y operar este tipo de dato.

El tipo cadena no existe. Se puede declarar una tabla de valores byte con longitud máxima igual a los caracteres a contener, más uno para el carácter 0 final que se añade a todas las cadenas.

Es importante notar que si una operación produce un valor negativo o se usa el operador menos unario (-) el resultado es el valor positivo complementado a dos. Por ejemplo:

```
byte b = 5;
word w;
b -= 9; // La expresión produce -4, pero el valor de b es 252
        // (256 - 4).
w = -8; // Produce 8, pero el valor de w es 65528 (65536 - 8).
```

Elementos

- Comentario de una línea: comienzan por //.
- Palabras clave: include, datafile, asm, define, byte, word, array, string, function, register, inline, using, if, else, do, while, for, repeat, goto, gosub and return. No se distinguen mayúsculas y minúsculas para las palabras claves ni para otros identificadores (etiquetas, variables, macros y funciones).
- Etiquetas: identificador que comienza por una letra de, seguido por una letra, dígito o carácter de subrayado.
- Expresiones.

Elementos de expresiones

En una expresión se pueden utilizar los siguientes elementos:

- **Paréntesis**: (y).
- **Operadores unarios**: mismo significado que en lenguaje C.

  ++ (pre y post operando)
  -- (pre y post operando)
  !
  -
  *
  ** (toma un valor word de la dirección especificada por el operando, misma prioridad que *)
  &
  ~

- **Operadores binarios**: mismo significado y uso que en lenguaje C.

  *
  /
  %
  +
  -

  <<
  >>

  <
  <=
  >
  >=
  ==
  !=

  &
  ^
  |

  &&
  ||

  *=
  /=
  %=
  +=
  -=
  <<=
  >>=
  &=
  ^=
  |=

- **Operador condicional**: ? y : (evalúa siempre las expresiones verdadera y falsa; es recomendable encerrar entre paréntesis las expresiones).
- **Variables byte**.
- **Variables word**.

- **Constantes byte**: en decimal o hexadecimal con el prefijo #. Por ejemplo 10 y #0A son el mismo valor.

- **Constantes de carácter**: equivalente a una constante byte, es un carácter encerrado entre comillas simples. Para especificar caracteres especiales es posible usar las secuencias de escape:

  \0 (para carácter ASCII 0)
  \a (para carácter ASCII 7)
  \b (para carácter ASCII 8)
  \e (para carácter ASCII 27)
  \f (para carácter ASCII 12)
  \n (para carácter ASCII 10)
  \r (para carácter ASCII 13)
  \t (para carácter ASCII 9)
  \v (para carácter ASCII 11)
  \' (para carácter ASCII 39)
  \" (para carácter ASCII 34)
  \\ (para carácter ASCII 92)
  \xnn (para carácter ASCII nn, donde nn está especificado en hexadecimal)

- **Constantes word**: en decimal o hexadecimal con el prefijo #. Por ejemplo 4096 y #1000 son el mismo valor.

- **Constantes de cadena**: una lista de caracteres encerrada entre comillas dobles. Es posible usar las mismas secuencias de escape que en las constantes de carácter.

- **Etiquetas**: corresponden al nombre de una tabla o a un punto en el código del programa, toma el valor donde la tabla se encuentra en memoria o la dirección del código donde la etiqueta se ha definido.

- **Funciones**: opcionalmente con parámetros, toma el valor de retorno de la función.

Todos los operadores con el mismo significado y prioridad que en lenguaje C.

Las variables de tabla no existen. Se puede implementar una tabla con punteros haciendo:

```
array byte tabla_byte[10]; // Declarara una tabla de valores
                           // byte de 10 elementos
byte n;
word i = 5; // Declara una variable word para índice de la tabla
n = *(tabla_byte + i); // Asigna a n el sexto elemento de la
                       // tabla
```

Si una tabla es de tipo word, la implementación del mismo ejemplo es:

```
array word tabla_word[10]; // Declara una tabla de valores word
                           // de 10 elementos
word n;
word i = 5; // Declara una variable word para índice de la tabla
n = **(tabla_word + i * 2); // Asigna a n el sexto elemento de
                            // la tabla
```

Para una tabla de tipo byte de dos dimensiones:

```
array word tabla_byte[40]; // Declara 40 elementos para una
                           // tabla de 4 filas x 10 columnas
byte n;
byte f = 2, c = 7; // Declara variables para índices de fila y
                   // columnas de la tabla
```

```
n = *(tabla _byte + f * 10 + c); // Asigna a n el octavo
                                 // elemento de la tercera fila
                                 // de la tabla
```

Y para una tabla de tipo word de dos dimensiones:

```
array word tabla_word[40]; // Declara 40 elementos para una
                           // tabla de 4 filas x 10 columnas
word n;
byte f = 2, c = 7; // Declara variables para índices de fila y
                   // columnas de la tabla
n = **(tabla_word + (f * 10 + c) * 2); // Asigna a n el octavo
                                       // elemento de la
                                       // tercera fila de la
                                       // tabla
```

Se puede simplificar la sintaxis utilizando macros. El primer ejemplo para una tabla unidimensional podría escribirse así:

```
array byte tabla_byte[10]; // Declarara una tabla de valores
                           // byte de 10 elementos
define datos(indice) = (*(tabla_byte + indice));
byte n;
word i = 5; // Declara una variable word para índice de la tabla
n = datos(i); // Asigna a n el sexto elemento de la tabla
```

Estructuras de control y sentencias

include <cadena>, ... ;

Inserta en el lugar donde se especifica los ficheros fuentes definidos por la lista de <cadena>. Un fichero se incluirá en el programa sólo una vez, si es incluido otra vez, es ignorado.

Ejemplos:

```
include "standard.ccz80";

include "msx.ccz80”, "graficos_msx.ccz80";
```

datafile <cadena> = <identificador>|<constante word>, … ;

Define al final del programa la etiqueta <identificador> y sitúa a continuación el contenido del fichero binario especificado por <cadena>, o bien sitúa dicho contenido en la dirección indicada por <constante word>.

Ejemplos:

```
datafile "DatosGraf.bin" = DatosGraficos;

datafile "Codigo.bin" = Codigo, “Datos.bin” = #A000;
```

asm { <cadena>, ... }

Inserta en el lugar donde se especifica las instrucciones ensamblador definidas por la lista de <cadena>.

Ejemplos:

```
asm { "ld a,5", "ld b,3", "add a,b" }

asm { "ld hl,0", "ld de,10", "ld b,7",
      "bucle:", "add hl,de", "djnz bucle" }
```

define <identificador>[(<identificador argumento>, …)] = <expresión>, … ;

Define la macro <identificador>, con los argumentos opcionales <identificador argumento> indicados entre paréntesis, como la <expresión> especificada, que debe usar todos los argumentos indicados. A partir de este punto del programa, todas las apariciones de <identificador> en el código serán sustituidas por <expresión>, en la que igualmente se sustituirán los nombres de los argumentos por los valores especificados como parámetros en el uso de la macro.

Ejemplos:

```
define Maximo = 100;

define Inicial = 5, Final = 35;

define Mayor(a, b) = (a > b ? a : b);
```

byte <identificador> [= <constante byte>|<constante carácter>], ... ;

Declara las variables byte especificadas por <identificador> (mismas reglas que para la definición de etiquetas), y opcionalmente define sus valores iniciales. Si el valor inicial no se define es cero.

Ejemplos:

```
byte n;

byte a = 3, b = 35, c = '\n';
```

word <identificador> [= <constante word>|<cadena>|<etiqueta>], ... ;

Declara las variables word especificadas por <identificador> (mismas reglas que para la definición de etiquetas), y opcionalmente define sus valores iniciales. Si el valor inicial no se define es cero.

Ejemplos:

```
byte n;

byte a = 3, b = 35, c = '\n';
```

array byte <identificador>[<constante word>], ... ;

Declara un espacio para una tabla de longitud <constante word> bytes. Se crea una etiqueta <identificador> con el valor del inicio de la tabla.

Ejemplos:

```
array byte tabla1[50];

array byte tabla2[10], tabla3[35];
```

array byte <identificador> = <cadena>, ... ;

Declara una tabla de valores byte con longitud igual a la longitud de <cadena> con los valores de cada carácter de <cadena> más un carácter cero al final. Se crea una etiqueta <identificador> con el valor del inicio de la tabla.

Ejemplos:

```
array byte texto1 = "hola";

array byte texto2 = "estoy", texto3 = "contento";
```

array byte <identificador> = { <constante byte>|<constante de carácter>, ... }, ... ;

Declara una tabla de valores byte con longitud igual al número de elementos encerrados entre llaves, con el valor de cada elemento de la lista. Se crea una etiqueta <identificador> con el valor de la dirección de inicio de la tabla.

Ejemplos:

```
array byte tabla1 = { 1, 3, 5, 7, 9 };

array byte tabla2 = { 'a', 'd', 'i', 'o', 's', 35, 0 },
           tabla3 = { #A1, #B3, #40, #16, #00 };
```

string <identificador>[<constante word>], ... ;

Equivalente a array byte <identificador>[<constante word> + 1], para mejor compresión del programa. Define una tabla de valores byte de longitud <constante word> + 1, para almacenar una cadena de un máximo de <constante word> caracteres.

Ejemplos:

```
string nombre[40];

string apellido1[25], apellido2[25];
```

string <identificador = <cadena>, ... ;

Equivalente a array byte <identificador> = <cadena>, para mejor compresión del programa. Define una tabla de valores byte de longitud igual a la longitud de <cadena> + 1, inicializándola con los valores ASCII de los caracteres de <cadena> más un carácter cero final.

Ejemplos:

```
string nombre = “Ana”;

string apellido1 = “López”, apellido2 = “Jiménez”;
```

array word <identificador>[<constante word>], ... ;

Declara espacio para una tabla de longitud <constante word> words (<constante word> * 2 bytes). Se crea una etiqueta <identificador> con el valor de la dirección de inicio de la tabla.

Ejemplos:

```
array word tabla1[30];

array word tabla2[5], tabla3[10];
```

array word <identificador> = { <constante word>|<cadena>|<etiqueta>, ... }

Declara una tabla de valores word de longitud igual al número de elementos encerrados entre llaves, con el valor de cada elemento de la lista. Se crea una etiqueta <identificador> con el valor de la dirección de inicio de la tabla.

Ejemplos:

```
array word tabla1 = { 1000, 32, 292 };

array word tabla2 = { "ccz80", texto1, texto2, 0 },
           tabla3 = { #FF00, #C000, #1000, #100, #10 };
```

function [register|inline] byte|word <identificador>(byte|word, ...) { <cadena>, ... }
[using <nombre función>, ...;]

Declara la función <identificador> con los parámetros especificados entre paréntesis (y) incluyendo las instrucciones en ensamblador indicadas por lista de <cadena> que se encuentren entre llaves { y }. Las funciones sólo pueden ser escritas en ensamblador y deben cargar al finalizar el valor de retorno en A (si la función de de tipo byte) o en HL (si la función es de tipo word).

La claúsula register es válida sólo para funciones de un solo parámetro y con ella el valor del parámetro no se introducirá en la pila, sino que llegará a la función en el registro A (si el parámetro es de tipo byte) o HL (si es de tipo word).

Si se especifica la cláusula inline la función no es llamada con CALL, sino que el código de la función se inserta directamente en el código del programa (como el código indicado en una sentencia asm). Una función inline debe tener uno o ningún argumento, no puede usar etiquetas y no puede ser especificada en una cláusula using de otra función. Si el argumento es un valor byte, éste es cargado en el registro A antes de la inserción del código de la función, si el argumento es un valor word éste es cargado en el par de registros HL. El resultado de la función debe cargarse al final del código en el registro A si la función devuelve un valor byte o en el par HL si la función devuelve un valor word.

Si se especifica la cláusula using las funciones indicada en la lista de <nombre función> se asocian a la función que se está declarando; así, si se usa en el programa esta función, su código se añadirá al código objeto, y también el código de las funciones de la lista de funciones indicadas tras using, incluso aunque éstas no se usen directamente en el programa. Esto es útil para definir funciones que usen código de otras funciones.

Ejemplos:

```
function byte suma_3(byte)
{ "ld ix,3", "add ix,sp", "ld a,(ix+0)", "add a,3", "ret" }

function register word suma_4(word)
{ “ld de,4”, “add hl,de”, “ret” }

function inline word multiplica_2(word) { "add hl,hl" }
```

if (<expresión>) { <sentencias 1> }|<sentencia 1> ;
[else { <sentencias 2> }|<sentencia 2> ;]

Evalúa <expresión> y ejecuta <sentencias 1> o <sentencia 1> si el valor de la expresión es verdadero (no cero) o si es falso (cero) se ejecuta <sentencias 2> o <sentencia 2>, si se especifica la parte else.

Ejemplos:

```
if (n > 5) n >>= 3;

if (n < 2) a = n * 2;
else a = n;
```

do { <sentencias> }|<sentencia> ;
while (<expresión>);

Ejecuta <sentencias> o <sentencia> mientras el valor de <expresión> sea verdadero (no cero), evaluando ésta al final de cada repetición.

Ejemplos:

```
do ++i;
while (i < 100);

do { n += maximo(a, b); n2 -= n; }
while (n != 0);
```

while (<expresión>) { <sentencias> }|<sentencia> ;

Ejecuta <sentencias> o <sentencia> mientras el valor de <expresión> sea verdadero (no cero), evaluando ésta al principio de cada repetición.

Ejemplos:

```
while (a == 0) a = valor_calculo(3, 6, b);

while (n == 1 && m == 3) { n = calculo(); m = calculo2(); }
```

for (<expresión 1>, ... ; <expresión 2> ; <expresión 3>, ...)
{ <sentencias> }|<sentencia>;

Evalúa la lista de <expresión 1> y mientras <expresión 2> es verdadero (no cero) ejecuta <sentencias> o <sentencia>, evaluando <expresión 2> antes de cada repetición y la lista de <expresión 3> tras cada repetición.

Ejemplos:

```
for (i = 0; i <= 100; ++i) n += i;

for (e = 128, v = 0, c = 0; c <= 8; e <<= 1, ++c)
  v += numero & e;
```

repeat (<expresión>) { <sentencias> }|<sentencia> ;

Ejecuta <sentencias> o <sentencia> el número de veces <expresión>. Para salir de un bucle repeat antes de su finalización se debe usar eliminar el valor del contador de la pila con una instrucción asm { “pop hl” } por ejemplo.

Ejemplos:

```
repeat (10) n += i;

repeat (3 * n + 2) { dibujar_punto(x, y); x += 2; y += 5; }
```

goto <etiqueta>;

Lleva el control del programa hasta la dirección especificada por <etiqueta>.

Ejemplos:

```
goto bucle_dibujo;
```

gosub <etiqueta>;

Lleva el control del programa hasta la dirección especificada por <etiqueta> hasta que una sentencias return devuelva el control a la instrucción siguiente a gosub.

Ejemplos:

```
gosub comprobar_choque;
```

return;

Devuelve el control del programa a la siguiente sentencia del último gosub ejecutado o finaliza el programa si no existe ningún gosub previo.

Ejemplos:

```
return;
```

<identificador>:

Declara la etiqueta <identificador> con el valor de la dirección de programa donde se encuentra definida.

Ejemplos:

```
rutina_inicializacion_pantalla:
```

<expresión>;

Evalúa la <expresión>.

Ejemplos:

```
++i;

n += 32 / t + funcion_calculo(t, 2);
```

Para las sentencias donde se requiera una constante byte o una constante word se puede utilizar una expresión formada por constantes byte, constantes de carácter, constantes word, paréntesis y todos los operadores válidos en ccz80 excepto los de asignación, siempre que la expresión produzca un valor entre -128 y 255 para una constante byte y entre -32768 y 65535 para una constante word.

Para las sentencia donde se requiera una constante de cadena se puede utilizar una expresión formada por varias cadenas concatenadas con el símbolo +.

Extensión del lenguaje

Es posible ampliar el lenguaje escribiendo nuevas funciones, genéricas o específicas para algún modelo de ordenador basado en Z80, utilizando su ROM particular, firmware y características particulares.

Siguiendo las indicaciones dadas para la sentencia function, se pude crear por ejemplo una función que recibe dos parámetros, el primero de tipo byte y el segundo de tipo word, devolviendo un valor word que es la suma de los dos parámetros recibidos:

```
function word suma_byte_y_word(byte, word)
```

```
{
  "ld ix,2",
  "add ix,sp ; IX apunta al último parámetro",
  "ld h,0",
  "ld l,(ix+3) ; HL toma el valor del primero parámetro (es un",
  "            ; valor byte y su valor está en el byte alto",
  "            ; del valor de la pila)",
  "ld d,(ix+1)",
  "ld e,(ix+0) ; DE toma el valor del segundo parámetro",
  "add hl,de ; HL = suma de primer y segundo parámetro",
  "ret ; El resultado debe quedar en HL porque la función es",
  "    ; tipo word"
}
```

Otro ejemplo para una función inline:

```
function inline word cambiar_signo_word(word)
// Cambia el signo del valor recibido como parámetro (devuelve 0
– parámetro)
{
  "; El valor del parámetro se encuentra ya en HL al ser una",
  "; función inline",
  "ld de,0",
  "ex de,hl",
  "or a ; Establecer CF a cero",
  "sbc hl,de ; El resultado se deja en HL porque la función de",
  "          ; de tipo word",
  "; No es necesaria la instrucción RET al ser una función",
  "; inline"
}
```

La misma función cambiar_signo_word sin cláusula inline:

```
function word cambiar_signo_word(word)
{
  "ld ix,2",
  "add ix,sp",
  "ld h,(ix+1)",
  "ld l,(ix+0) ; HL toma el valor del parámetro",
  "ld de,0",
  "ex de,hl",
  "or a",
  "sbc hl,de ; El resultado se deja en HL",
  "ret"
}
```

Sintaxis del ensamblador

La sintaxis para escribir el código ensamblador de las funciones y de las sentencias asm está basada en el ensamblador GENA del paquete DEVPAC. Las principales reglas del ensamblador integrado en el compilador ccz80 son:

- Acepta todas las instrucciones incluyendo las indocumentadas usando los registros IXh, IXl, IYh, IYl y las de rotación/desplazamiento.
- La declaración de etiquetas y símbolos con EQU necesita dos puntos (:) tras el nombre.
- La longitud del nombre de las etiquetas y símbolos es ilimitada.
- Las constantes numéricas pueden indicarse en decimal, hexadecimal (con el prefijo #) y en binario (con el prefijo %).

- Las constantes de cadena y constantes de carácter se delimitan con comillas dobles. No se deben usar secuencias de escape en cadenas o constantes de carácter.
- Las expresiones pueden usar constantes numéricas, constantes de carácter, símbolos o etiquetas y los operadores + (suma), - (resta), * (producto), / (división entera), ? (módulo), & (y lógico), @ (o lógico) y ! (xor lógico). También se puede usar el símbolo $ para especificar la dirección de la instrucción actual.
- Las directivas permitidas son ORG, EQU, DEFB, DEFW, DEFM y DEFS.
- Las directivas condicionales IF, ELSE y END no se aceptan.
- No se diferencian mayúsculas y minúsculas para instrucciones, nombres de registros, etiquetas y símbolos.
- Los comandos de ensamblador (*E, *H, *S, …) no se reconocen.
- Por supuesto, los números de línea no están permitidos.

Librería estándar

La librería de funciones estándar se encuentra en el fichero standard.ccz80. Se debe incluir la librería en el programa antes de usar cualquiera de sus funciones (normalmente al principio del código fuente) con la siguiente sentencia:

```
include "standard.ccz80";
```

Las funciones que incluye son:

Funciones de manipulación de cadenas

strlen(<cadena>): Devuelve la longitud de <cadena>.

strcpy(<cadena 1>, <cadena 2>): Copia <cadena 2> en <cadena 1>. Devuelve la dirección de <cadena 1>.

strncpy(<cadena 1>, <cadena 2>, <n>): Copia los primeros <n> caracteres de <cadena 2> en <cadena 1>. Devuelve la dirección de <cadena 1>.

strcat(<cadena 1>, <cadena 2>): Añade <cadena 2> al final de <cadena 1>. Devuelve la dirección de <cadena 1>.

strset(<cadena>, <carácter>, <n>): Almacena en <cadena> el <carácter> repetido <n> veces. Devuelve la dirección de <cadena>.

strltrm(<cadena>, <carácter>): Borra el <carácter> si se encuentra al principio de <cadena>.

strrtrm(<cadena>, <carácter>): Borra el <carácter> si se encuentra al final de <cadena>.

strlpad(<cadena>, <n>, <carácter>): Rellena <cadena> por la izquierda con el <carácter> hasta que tenga longitud <n>.

strrpad(<cadena>, <n>, <carácter>): Rellena <cadena> por la derecha con el <carácter> hasta que tenga longitud <n>.

strupr(<cadena>): Convierte a mayúsculas todos los caracteres de <cadena>. Devuelve la dirección de <cadena>.

strlwr(<cadena>): Convierte a minúsculas todos los caracteres de <cadena>. Devuelve la dirección de <cadena>.

strcmp(<cadena 1>, <cadena 2>): Devuelve 0 si <cadena 1> es igual a <cadena 2>, 1 si <cadena 1> es menor alfabéticamente que <cadena 2> o 2 si <cadena 1> es mayor alfabéticamente que <cadena 2>.

strchr(<cadena>, <carácter>): Devuelve la dirección de memoria donde <carácter> se encuentra en <cadena> o devuelve 0 si no lo encuentra.

strstr(<cadena 1>, <cadena 2>): Devuelve la dirección de memoria donde <cadena 2> se encuentra en <cadena 1> o devuelve 0 si no se encuentra.

Funciones de conversión

btoa(<cadena>, <valor>): Genera en <cadena> la representación decimal de <valor>. Devuelve la dirección de <cadena>.

btoh(<cadena>, <valor>): Genera en <cadena> la representación hexadecimal de <valor>. Devuelve la dirección de <cadena>.

bton(<cadena>, <valor>): Genera en <cadena> la representación binaria de <valor>. Devuelve la dirección de <cadena>.

wtoa(<cadena>, <valor>): Genera en <cadena> la representación decimal de <valor>. Devuelve la dirección de <cadena>.

wtoh(<cadena>, <valor>): Genera en <cadena> la representación hexadecimal de <valor>. Devuelve la dirección de <cadena>.

wton(<cadena>, <valor>): Genera en <cadena> la representación binaria de <valor>. Devuelve la dirección de <cadena>.

atob(<cadena>): Devuelve como byte el valor de <cadena> considerando que contiene un valor representado en decimal.

htob(<cadena>): Devuelve como byte el valor de <cadena> considerando que contiene un valor representado en hexadecimal.

ntob(<cadena>): Devuelve como byte el valor de <cadena> considerando que contiene un valor representado en binario.

atow(<cadena>): Devuelve como word el valor de <cadena> considerando que contiene un valor representado en decimal.

htow(<cadena>): Devuelve como word el valor de <cadena> considerando que contiene un valor representado en hexadecimal.

ntow(<cadena>): Devuelve como word el valor de <cadena> considerando que contiene un valor representado en binario.

Funciones de carácter

toupper(<carácter>): Devuelve el <carácter> en mayúsculas.

tolower(<carácter>): Devuelve el <carácter> en minúsculas.

Funciones de números aleatorios

srand(<valor>): Define <valor> como semilla para de los siguientes números aleatorios.

rand(): Devuelve un valor numérico entre 0 y 65535.

previousrand(): Devuelve el último valor numérico generado.

Funciones de entrada/salida de puertos

in(<puerto>): Devuelve el valor leído desde el <puerto>.

out(<puerto>, <valor>): Envía <valor> al <puerto>.

Funciones matemáticas

absb(<valor>): Devuelve el valor absolute de <valor> como byte.

absw(<valor>): Devuelve el valor absolute de <valor> como word.

sgnb(<valor>): Devuelve -1, 0 ó 1 según <valor> como byte sea negativo, cero o positivo.

sgnw(<valor>): Devuelve -1, 0 ó 1 según <valor> como word sea negativo, cero o positivo.

min(<valor1>, <valor2>): Devuelve el menor valor de los dos dados.

max(<valor1>, <valor2>): Devuelve el mayor valor de los dos dados.

sqrt(<valor>): Devuelve la raíz cuadrada entera de <valor>.

power(<base>, <exponente>): Calcula el resultado de elevar <base> a <exponente>.

Otras funciones

exitrepeat(<dirección>): Salta a <dirección> desde dentro de un bucle repeat hasta fuera de él.

signedword(<valor>): Convierte el <valor> byte a word considerando el signo.

endprogram(): Devuelve la dirección del último byte utilizado por el programa, para considerar memoria libre tras él.

Ejemplos del lenguaje

Ejemplo 1. Calcula la suma de los números 1 a 100:

```
word suma = 0; // Resultado de la suma
byte i = 1;
while (i <= 100) suma += i++;
printw(s); // printw debe ser una función para imprimir un valor
word en pantalla
return; // Finaliza programa
```

Ejemplo 2. Encuentra el máximo valor en una tabla de valores word:

```
array word lista = { 50, 225, 32, 450, 1003, 32, 9, 8, #FFFF };
// Valor #FFFF es la marca de final de tabla
word p; // Puntero para recorrer la tabla
word maximo = 0;
for (p = lista; **p != #FFFF; p += 2) // p es un puntero a
valores word, por eso se incrementa 2 posiciones para acceder al
siguiente elemento
  if (maximo == 0) maximo = **p;
  else if (**p > maximo) maximo = **p;
printw(maximo); // printw debe ser una función para imprimir un
valor word en pantalla
return; // Finaliza programa
```

Ejemplo 3. Evalúa algunos productos y suma los resultados:

```
byte a, b, resultado = 0;
a = 5;
b = 3;
gosub CalcularProducto; // 5 * 3
a = 1;
b = 4;
gosub CalcularProducto; // 1 * 4
a = 7;
++b;
gosub CalcularProducto; // 7 * 5
printw(resultado); // resultado = 15 + 4 + 35 = 54
return; // Finaliza programa
// Subrutina CalcularProducto
// Recibe en a y b los operandos y acumula en resultado el valor
de la multiplicación
CalcularProducto:
resultado += a * b;
return; // Vuelve de subrutina CalcularProducto
```