

CPCBasic

Compilador cruzado para Amstrad CPC

Descripción.....	2
Sintaxis del lenguaje.	2
Constantes.	2
Tipos de datos del lenguaje.	3
Palabras clave: directivas.....	3
REM / '.....	3
CHAIN.....	3
DEFBYTE, DEFINT, DEFLNG, DEFREAL, DEFSTR.....	3
DATA.....	3
CONST.....	4
DIM.....	4
DEF FN.....	4
DEF SUB.....	5
Palabras clave: instrucciones.....	5
Asignación.....	5
RESTORE.....	5
READ.....	5
GOSUB.....	5
RETURN.....	5
GOTO.....	6
Etiqueta.....	6
IF ... THEN ... ELSE.....	6
FOR ... TO ... STEP.....	6
NEXT.....	6
WHILE.....	6
WEND.....	7
END.....	7
Programación en ensamblador.....	7

Descripción.

CPCBasic es un compilador cruzado que funciona en modo de línea de Windows, escrito en C#, que genera código binario para ejecutar en Amstrad CPC o emuladores. El compilador consta de un solo archivo llamado CPCBasic.exe, y es necesario tener instalado el paquete Microsoft .NET Framework para su correcto funcionamiento.

Para utilizar el compilador CPCBasic se usa la siguiente sintaxis:

```
CPCBasic <archivo fuente> [/464 | /664 | /6128] [/ORG=<dirección>]
```

donde <archivo fuente> es el fichero de texto que contiene el código fuente a compilar. Puede indicarse una de las opciones /464, /664 ó /6128 para definir el modelo de CPC para el que se desea compilar (se tomará /464 si no se indica ninguna, siendo /664 y /6128 equivalentes). También puede especificarse la opción /ORG para establecer el valor dado por <dirección> (con el formato establecido para las constantes numéricas en CPCBasic) como inicio del código objeto generado (se tomará &H4000 si no se especifica).

El resultado de la compilación es un programa ensamblador resultado de la traducción del código fuente en CPCBasic. Por tanto será necesario disponer también de un ensamblador para generar el código binario final. Son compatibles con el código generado el ensamblador pasmo, el ensamblador del emulador WinCPC y el ensamblador DEVPAC bajo AMSDOS, entre otros.

El programa devuelve en valor para ERRORLEVEL igual a 0 si no se ha producido ningún error, y un valor igual a 1 si se ha producido cualquier error.

Sintaxis del lenguaje.

El compilador trabaja con un lenguaje de programación basado en el Locomotive Basic original de los Amstrad CPC. Este lenguaje está basado en un conjunto de palabras claves al que se le pueden añadir funciones y procedimientos definidos por el usuario.

Incluye como mejoras principales la existencia de constantes, más tipos de datos, uso de etiquetas en lugar de números de línea (suprimiendo éstos) y la posibilidad de escribir una sentencia en diferentes líneas del fichero fuente utilizando el carácter subrayado (_) al final de cada línea que se quiera continuar en la siguiente.

Cualquier elemento que se quiera utilizar, excepto etiquetas, debe estar definido previamente. La declaración de constantes, variables, funciones y órdenes se puede hacer en cualquier punto del programa, previo a su utilización.

No se hace distinción entre mayúsculas y minúsculas.

Constantes.

Se pueden utilizar los siguientes tipos de constantes:

- Números decimales: formados por dígitos 0-9, pudiendo tener el signo + ó – al principio y . para separar la parte entera de la fraccional. Siempre deben comenzar por signo o un dígito.
- Números hexadecimales: formados por dígitos 0-9 y letras A-Z, con el prefijo &H ó &.
- Números binario: formados por dígitos 0-1, con el prefijo &X.
- Cadenas de texto: formadas por hasta 255 caracteres de longitud, encerradas entre comillas (").

Tipos de datos del lenguaje.

Los identificadores del lenguaje son nombres que comienzan por una letra A-Z (sólo símbolos del alfabeto inglés) seguido de cualquier cantidad de otras letras, dígitos 0-9 ó el símbolo subrayado ().

Los identificadores de datos (constantes, variables y funciones) deben llevar un sufijo que indica qué tipo de dato contienen o devuelven. Este sufijo de tipo puede suprimirse si el identificador es del tipo definido por defecto en el momento de su uso según su primera letra (ver palabras clave DEFBYTE, DEFINT, DEFLNG, DEFREAL, DEFSTR).

Los tipos de datos son:

- Byte, con sufijo #, para números de 8 bits con valores de -128 a +127.
- Int, con sufijo %, para números de 16 bits con valores de -32768 a +32767.
- Long, con sufijo &, para números de 32 bits con valores de -2147483648 a +2147483647.
- Real, con sufijo !, para números en coma flotante de 5 bytes de capacidad. El rango de valores que puede tomar es de -1E+38 a +1E+38.
- Cadena de texto, con sufijo \$, para textos de hasta 255 caracteres de longitud.

Palabras clave: directivas.

Estas palabras clave se evalúan durante la compilación, no importando si están situadas dentro de partes de código que no se realizarán nunca por parte del programa.

REM / ‘

Permite escribir un comentario en el programa fuente. Se puede utilizar REM o el carácter ‘ y todo el texto existente tras ellos no se compilará. Si se utiliza REM debe separarse de la instrucción previa mediante dos puntos (:), si se utiliza ‘ no es necesaria la separación.

Sintaxis: REM <texto>

Ejemplo: REM Inicio de operaciones

CHAIN

Permite incluir en el punto donde se utilice esta directiva el código fuente que se encuentre en los archivos especificados.

Sintaxis: CHAIN <lista de: <nombre de archivo>>

Ejemplo: CHAIN "FUNCIONES.BAS", "ORDENES.BAS"

DEFBYTE, DEFINT, DEFLNG, DEFREAL, DEFSTR

Especifica que el tipo por defecto que se aplicará a las constantes, variables y funciones declaradas y utilizadas en adelante será byte, int, long, real o cadena de texto respectivamente, si su nombre comienza por una letra incluida en el rango especificado.

Si no se ha especificado ninguna sentencia de este grupo el tipo por defecto para cualquier identificado que comience por cualquier letra es real.

Sintaxis: DEFBYTE <lista de: <letra inicial>[-<letra final>]>

Ejemplo: DEFBYTE B, M-Q

DATA

Permite definir datos en memoria para su obtención durante el programa mediante READ. Para cada dato es obligatorio especificar su tipo utilizando los sufijos de tipo de dato, excepto para

los datos de tipo cadena de texto que se especificarán entre comillas dobles ("). El límite de cada valor, según el sufijo que se indique es el correspondiente a ese tipo de datos.

Es imprescindible evitar que el flujo del programa llegue hasta una sentencia DATA.

Sintaxis: DATA <lista de: <valor> <sufijo tipo dato>>

Ejemplo: DATA 1#, 2%, 3&, 4.5!, "texto"

CONST

Declara una constante de un determinado tipo para poderla utilizar posteriormente en cualquier expresión.

Sintaxis: CONST <lista de: <<nombre>[<tipo>] = <valor constante>>

Ejemplo: CONST NOMBRE\$ = "Programa", INICIAL! = 0.25, MAXIMO& = 75000

DIM

Declara una variable, individual o como tabla. Los valores iniciales que toman las variables son cero para datos numéricos o cadenas de longitud cero para datos de tipo cadena.

Si la variable es de tipo tabla el índice del primer elemento de cada dimensión es cero y el último es el indicado en la declaración.

Sintaxis: DIM <lista de: <<nombre>[<tipo>] [(<lista de: <dimensiones>>)]>>

Ejemplo: DIM A\$, LINEA\$(25), CARACTERES\$(40, 25)

DEF FN

Permite definir funciones de usuario que deben escribirse en lenguaje ensamblador, pudiendo utilizar las características y sintaxis del programa ensamblador que se utilizará posteriormente para generar el código binario. Cada función debe tener un nombre con las características indicadas para el identificador de dato, incluyendo el sufijo de tipo si no se quiere utilizar el tipo de defecto en el instante de la definición. Estas funciones pueden recibir parámetros y deben devolver un valor. Los parámetros se reciben por valor en la pila y se encuentran en orden inverso a como se especificaron en la llamada a la función. El valor del resultado devuelto debe dejarse en la memoria y hacer que al finalizar el código de la función el registro HL contenga la dirección del resultado.

En la definición de la función los parámetros se indican y pueden indicar valores por defecto para cada parámetro, de tal forma que si en la llamada a la función alguno se omite se usará dicho valor. Las instrucciones de ensamblador que forman la función se indicarán como una concatenación de cadenas de texto.

El código que constituye la definición de una función de usuario se vuelca al resultado de la compilación tal cual, y será el ensamblador utilizado posteriormente el que examinará este código.

En el momento de utilizar una función en una expresión es imprescindible escribir los paréntesis tras su nombre, incluso aunque éstos no contengan ningún parámetro.

Sintaxis: DEF FN <nombre>[<identificador de tipo>]([<lista de: <sufijo tipo dato [= <valor defecto>]>>]) = <cadena de texto> + <cadena de texto> + ...

Ejemplo: DEF FN SUMA#(#, # = 1) = "LD A,1" + "LD HL,&FFFF" + "CALL _RESERVAR_HEAP" + "PUSH HL" + "LD IX,2" + "ADD IX,SP" + "LD H,(IX+3)" + "LD L,(IX+2)" + "LD D,(IX+1)" + "LD E,(IX+0)" + "LD A,(DE)" + "ADD (HL)" + "POP HL" + "LD (HL),A" + "RET"

DEF SUB

Permite definir órdenes de usuario que deben escribirse en lenguaje ensamblador. Ver DEF FN para conocer los detalles sobre nombres de órdenes, parámetros, valores por defecto, etc. Una orden de usuario no devuelve ningún valor ni debe indicarse ningún tipo de dato en su definición.

Sintaxis: DEF SUB <nombre ([<lista de: < sufijo tipo dato [= <valor defecto>]>>]) = <cadena de texto> + <cadena de texto> + ...

Ejemplo: DEF SUB PRINT_CHAR(# = &20) = "LD IX,2" + "ADD IX,SP" + "LD H,(IX+1)" + "LD L,(IX+0)" + "LD A,(HL)" + "CALL &BB5A" + "RET"

Palabras clave: instrucciones.

Estas sentencias se realizan sólo si el flujo del programa llega hasta ellas.

Asignación

Esta instrucción no corresponde a ninguna palabra clave. Se trata de la operación de asignar a una variable un determinado valor producido por una expresión formado por operandos (constantes, variables, llamadas a funciones) y operadores (+, -, *, /, \, MOD, ^, AND, OR, XOR, NOT y @) o bien por el nombre de una etiqueta para obtener su dirección en memoria.

Sintaxis: <variable> = <expresión>

Ejemplo: N1! = -B! + SQR!(4 * A! * C#) / 2 * A! : A\$ = C\$ + FIN_LINEA\$: N% = ETIQUETA_1

RESTORE

Estable la sentencia DATA que sigue a la etiqueta indicada para que las siguientes sentencias READ lean los datos especificados en ella.

Sintaxis: RESTORE <etiqueta>

Ejemplo: RESTORE DATOS_INICIALIZACION

READ

Permite asignar a variables los datos declarados con DATA. Las variables que se especifiquen para asignar los datos deben ser del mismo tipo que los datos declarados y leerse en el mismo orden. Antes de utilizar READ es imprescindible utilizar la directiva RESTORE para inicializar el lugar a partir del cual se leerán los datos declarados.

Sintaxis: READ <lista de: <variable>>

Ejemplo: READ NOMBRE\$(N), CODIGO&(N)

GOSUB

Permite realizar la subrutina que comienza a partir de la etiqueta especificada. Esta subrutina debe finalizar con la sentencia RETURN.

Sintaxis: GOSUB <etiqueta>

Ejemplo: GOSUB COMPROBAR_NUMERO

RETURN

Da por finalizada la realización de una subrutina haciendo que el flujo del programa regrese a la sentencia siguiente al GOSUB que inició la subrutina donde se especifique RETURN.

Sintaxis: RETURN
Ejemplo: RETURN

GOTO

Lleva el flujo del programa a la instrucción siguiente a la etiqueta especificada.

Sintaxis: GOTO <etiqueta>
Ejemplo: GOTO INICIAR_CALCULO

Etiqueta

Define mediante un identificador la posición en el programa de un punto al que se puede acceder mediante GOTO o GOSUB. También especifica la posición a partir de la que se tomarán datos para READ o un espacio de memoria para indicar en las órdenes o funciones donde sea necesario.

Sintaxis: <etiqueta>
Ejemplo: INICIO

IF ... THEN ... ELSE

Evalua una expresión y si es cierta realiza las instrucciones especificadas o si no es cierta realiza otras instrucciones.

Sintaxis: IF <expresión> THEN <sentencias> [ELSE <sentencias>]
Ejemplo: IF A>3 OR B=0 THEN GOTO REPETIR ELSE A=0:B=1

FOR ... TO ... STEP ...

Inicial un bucle asignando a una variable un valor inicial hasta que ésta tome el valor final, incrementando o decrementandola según el valor indicado. Si no se especifica valor de incremento/decremento éste será 1. El bucle realizará todas las sentencias incluidas entre FOR y NEXT.

Sintaxis: FOR <variable> = <valor inicial> TO <valor final> [STEP <valor incremento>]
Ejemplo: FOR N = 1 TO 10 STEP 2

NEXT

Finaliza un bucle iniciado por FOR.

Sintaxis: NEXT
Ejemplo: NEXT

WHILE

Inicia un bucle que se realizará mientras la expresión que se indica sea cierta. El bucle realizará todas las sentencias entre WHILE y WEND.

Sintaxis: WHILE <expresión>
Ejemplo: WHILE A<10

WEND

Finaliza un bucle iniciado con WHILE.

Sintaxis: WEND

Ejemplo: WEND

END

Finaliza el programa, regresando al BASIC.

Programación en ensamblador.

Para facilitar la programación de las funciones y órdenes definidas por el usuario se pueden utilizar las rutinas que se incluyen en cualquier programa CPCBasic. En el código ensamblador que se genera tras una compilación se pueden ver estas rutinas con una descripción al principio de cada una de ellas de su función, valores de entrada y de salida.

Para la programación en ensamblador hay que tener en cuenta que existe en memoria un espacio dinámico (llamado heap) donde se puede reservar los bytes necesarios para almacenar valores. Cada dato que se almacena en el espacio dinámico se compone de 3 bytes de cabecera seguidos de tantos bytes como se necesiten para almacenarlo. Los dos primeros bytes de la cabecera es la dirección de la variable cuyo valor está almacenado en ese espacio dinámico (para cadenas de texto) o bien el valor &FFFF que indica que es un espacio temporal que se liberará al finalizar el código correspondiente a la sentencia actual. El tercer byte de la cabecera es la longitud que ocupan los datos del valor almacenado.

Para las variables numéricas declaradas en el programa CPCBasic se reserva el espacio necesario identificado con una etiqueta `_NOMBRE_TIPO`, donde NOMBRE es el nombre dado en el código fuente, sin sufijo de tipo, y TIPO es BYTE, INT, LNG o REAL según el tipo de variable.

Las variables de cadena tienen reservado dos bytes, identificados por una etiqueta `_NOMBRE_STR`, para indicar cual es su dirección en el espacio dinámico. Esta dirección indica el comienzo de los caracteres de la cadena. La longitud de la cadena se encuentra en el byte precedente a esta dirección (tercer byte de cabecera).