

# ABASC: MANUAL DEL USUARIO

## Un compilador cruzado de BASIC para los Amstrad CPC

- Introducción
  - Influencias
  - Un recorrido por las distintas versiones de Locomotive BASIC
    - Versión 1.0
    - Versión 1.1
    - Versión 2
    - Versión 2 Plus
- Referencias
- Sintaxis soportada por ABASC
  - Ejemplo 1 (sintaxis compatible con BASIC 1.0 y 1.1)
  - Ejemplo 2 (sintaxis incluyendo varias de las mejoras de BASIC 2)
- Herramientas adicionales
- Uso del compilador
  - Opciones
  - Creación de un proyecto usando BASPRJ
- Peculiaridades del compilador
  - Tipos y variables
    - Cadenas de texto
    - Arrays
    - Estructuras con RECORD
  - Procedimientos y Funciones
  - Uso de código ensamblador
  - Punteros
  - Gestión de la memoria
  - Uso del Firmware
  - Librerías
- Comandos y sintaxis del lenguaje
  - Notación
  - Listado de comandos y funciones
    - ABS(<numeric expression>)
    - AFTER delay[,timer] GOSUB etiqueta
    - ASC(string)
    - ASM string[,string]\*
    - ATN(x)
    - AUTO linenumber[,increment]
    - BIN\$(number,digits)
    - BORDER colour1[,colour2]
    - CALL address[,list of parameters]
    - CAT
    - CHAIN
    - CHAIN MERGE string
    - CHR\$(x)
    - CINT(x)
    - CLEAR

- CLEAR INPUT
- CLG [tinta]
- CLOSEIN
- CLOSEOUT
- CLS [#x]
- CONST
- CONT
- COPYCHR\$(#canal)
- COS(x)
- CREAL(x)
- CURSOR sistema[,usuario]
- DATA lista-de-constantes
- DECLARE variable[\$ FIXED longitud]),...
- DEC\$(numero,patron)
- DEF FN nombre(parametros)=expresion
- DEFINT, DEFSTR, DEFREAL
- DEG
- DELETE bajo-alto
- DERR
- DI
- DIM array(indice1, indice2, ...) [FIXED longitud]
- DRAW x,y[,i[,modo]]
- DRAWR x,y[,i[,modo]]
- EDIT linea[-linea]
- EI
- END
- END FUNCTION
- END SUB
- ENT numero de envolvente, secciones
- ENV número de envolvente, secciones
- EOF
- ERASE arrayname
- ERL
- ERR
- ERROR integer
- EVERY tiempo[,temporizador] GOSUB etiqueta
- EXIT FOR
- EXIT WHILE
- EXP(x)
- FILL
- FIX(x)
- FOR variable=inicio TO fin STEP variacion
- FRAME
- FRE(x)
- FUNCTION nombre(parametros) [ASM]
- GOSUB etiqueta
- GOTO etiqueta
- GRAPHICS PAPER tinta
- GRAPHICS PEN tinta,modo
- HEX\$(x,digitos)
- HIMEM
- IF expression THEN expression ELSE expression END IF
- INK tinta,color1[,color2]
- INKEY(tecla)

- INKEY\$
- INP(puerto)
- INPUT [#canal,] ["mensaje"] [;]variable1,variable2...
- INSTR([posición,]cadena1,cadena2)
- INT(x)
- JOY(joystick)
- KEY tecla,cadena
- KEY DEF tecla,repertir[,<normal>[,<mayus>[,<control>]]]
- LABEL etiqueta
- LEFT\$(cadena,n)
- LEN(cadena)
- LET variable=expression
- LINE INPUT [#canal,] [;] [cadena;] <variable>
- LIST [rango de líneas] [, #canal]
- LOAD fichero[,dirección]
- LOCATE [#canal,]x,y
- LOG(x)
- LOG10(x)
- LOWER\$(cadena)
- MASK mascara[,puntoinicial]
- MAX(a,b[,c,d,e...])
- MEMORY maxdir
- MERGE fichero
- MID\$(cadena,inicio[,n])
- MIN(a,b[,c,d,e,f...])
- MODE n
- MOVE x,y[tinta[,modo]]
- MOVER x,y[,tinta[,modo]]
- NEW
- NEXT variable
- ON n GOSUB lista de etiquetas
- ON n GOTO lista de etiquetas
- ON BREAK GOSUB etiqueta
- ON BREAK STOP
- ON ERROR GOTO etiqueta
- ON SQ (canal) GOSUB etiqueta
- OPENIN fichero
- OPENOUT fichero
- ORIGIN x,y[,izq,der,arriba,abajo]
- OUT puerto,n
- PAPER [#canal,]tinta
- PEEK(direccion)
- PEN [#canal,]tinta
- PI
- PLOT x,y[,tinta[,modo]]
- PLOTR x,y[,tinta[,modo]]
- POKE direccion,n
- POS(#canal)
- PRINT [#canal,][lista de elementos]
- RAD
- RANDOMIZE [n]
- READ lista-de-variables
- READIN lista-de-variables
- RECORD nombre;lista-de-variables

- RELEASE canal
- REM texto
- REMAIN(temporizador)
- RENUM nueva-línea, línea-orígen, incremento
- RESTORE [etiqueta]
- RESUME
- RETURN
- RIGHT\$(cadena,n)
- RND[(0)]
- ROUND(x[,n])
- RUN [etiqueta | fichero]
- SAVE fichero[,tipo][,dirección,tamaño[,entrada]]
- SGN(x)
- SHARED variable | array [,variable | array]
- SIN(x)
- SOUND canal,período-ono,duración,volumen,env,ent,ruido
- SPACE\$(n)
- SPEED INK t1,t2
- SPEED KEY espera,repeticón
- SPEED WRITE n
- SQ canal
- SQR(x)
- STOP
- STR\$(x)
- STRING\$(n,carácter)
- SUB [(parámetros)] [ASM]
- SYMBOL carácter,valor1,valor2,...,valor8
- SYMBOL AFTER n
- TAG [#canal]
- TAGOFF [#canal]
- TAN(x)
- TEST(x,y)
- TESTR(x,y)
- TIME[(n)]
- TROFF
- TRON
- UNT(n)
- UPPER\$(cadena)
- VAL(cadena)
- VPOS(#canal)
- WAIT puerto,máscara[,inversión]
- WEND
- WHILE condición
- WIDTH n
- WINDOW [#canal,líq,derecha,arriba,abajo]
- WINDOW SWAP canal1,canal2
- WRITE [#canal],dato1,dato2,...
- XPOS
- YPOS
- ZONE n
- Apéndice I: Depurar programas compilados
  - Comprobación del código BASIC
  - Depuración paso a paso de nuestro código
- Apéndice II: Ampliando el compilador

# Introducción

**ABASC (BASic Compiler)** es un compilador cruzado escrito íntegramente en Python y sin dependencias externas, lo que favorece su portabilidad a cualquier sistema que disponga de una instalación estándar de **Python 3**.

Está diseñado para soportar el dialecto de BASIC creado por **Locomotive Software** para los microordenadores Amstrad CPC, de modo que toda la documentación existente sobre este lenguaje siga siendo plenamente relevante y útil.

Además, al tratarse de un compilador cruzado que se ejecuta en sistemas modernos, ABASC incorpora diversas características de **Locomotive BASIC 2 Plus**, lo que permite una experiencia de desarrollo más cercana a los lenguajes actuales sin renunciar al estilo clásico del BASIC original.

## Influencias

ABASC debe su existencia al compilador de BASIC **CPCBasic**  
<https://cpcbasic.webcindario.com/CPCBasicSp.html>. Probablemente, ABASC no existiría si el proyecto siguiese activo y sus fuentes fueran públicas y accesibles.

## Un recorrido por las distintas versiones de Locomotive BASIC

### Versión 1.0

La primera versión de este BASIC apareció con los Amstrad CPC 464. Era un lenguaje relativamente rápido en comparación con otros BASIC de la época. Entre sus ventajas principales contaba con un acceso amplio a las funcionalidades del chip de audio. Utilizaba números de línea como etiquetas para las sentencias GOTO y GOSUB.

### Versión 1.1

Introducida con los CPC 664 y 6128, esta versión corregía diversos fallos e incorporaba nuevas funciones, como FRAME, COPYCHR\$ o FILL. Aun así, seguía requiriendo el uso de números de línea.

### Versión 2

Lanzada en 1987 para los Amstrad PC 1512 y 1640, esta versión eliminaba la necesidad de numerar líneas gracias al comando LABEL y permitía crear aplicaciones para el entorno gráfico GEM, aunque todavía no ofrecía mecanismos avanzados de estructuración del código.

### Versión 2 Plus

Aparecida en 1989, esta revisión añadía FUNCTION, SUB, sentencias IF de varias líneas y otras mejoras orientadas a facilitar el desarrollo de programas más estructurados.

# Referencias

Este manual no trata de ser una guía exhaustiva de programación en BASIC. Como material de consulta sobre la programación en Locomotive BASIC es más recomendable consultar los siguientes textos:

- Amstrad CPC464 - Manual del Usuario (I. Spital, R. Perry, W. Poel and C. Lawson)
- Manual de Referencia BASIC para el Programador (Amsoft)
- Amstrad CPC6128 - Manual del Usuario (I. Spital, R. Perry, W. Poel and C. Lawson)
- BASIC 2 User Guide (Locomotive Software Ltd.)
- BASIC 2 PLUS Language Reference (Locomotive Software Ltd.)
- Using Locomotive BASIC 2 on the Amstrad 1512 (Robert Ransom)

Para ampliar conocimientos sobre el Firmware del Amstrad CPC464 y CPC6128, o sobre programación en ensamblador para el procesador Z80, se recomiendan los siguientes libros de consulta:

- CPC464/664/6128 FIRMWARE, ROM routines and explanations (B. Godden, P. Overell, D. Radisic)
  - The Amstrad CPC Firmware Guide (Bob Taylor)
  - Z80 Assembly Language Programming (Lance A. Leventhal)
  - Ready Made Machine Language Routines For the Amstrad CPC (Joe Pritchard)
  - Código máquina para principiantes con Amstrad (Steve Kramer)
- 

## Sintaxis soportada por ABASC

1. No es necesario usar números de línea.
2. Se pueden definir etiquetas para saltos mediante LABEL.
3. Bloques IF ... THEN ... ELSE ... END IF de varias líneas.
4. Definición de procedimientos con FUNCTION y SUB.
5. Inclusión de código ensamblador mediante ASM.
6. Inclusión de código BASIC externo con CHAIN MERGE.
7. Definición de estructuras de datos con RECORD.

### Ejemplo 1 (sintaxis compatible con BASIC 1.0 y 1.1)

```
10 MODE 1
20 BORDER 0
30 PAPER 3
40 INK 0,1,2
50 PEN 0
60 PRINT "Hello world"
70 END
```

### Ejemplo 2 (sintaxis incluyendo varias de las mejoras de BASIC 2)

```
RECORD person; name$ FIXED 10, age, birth
DIM records$(5) FIXED 14
```

```

CLS
FOR I=0 TO 5
    READ records$(i).person.name$, records$(i).person.age,
records$(i).person.birth
    PRINT "Customer:", records$(i).person.name$
    PRINT "Age:", records$(i).person.age
    PRINT "Born in:", records$(i).person.birth
NEXT
END

DATA "Xavier", 49, 1976
DATA "Ross", 47, 1978
DATA "Gada", 12, 2013
DATA "Anabel", 51, 1974
DATA "Rachel", 45, 1980
DATA "Elvira", 20, 2005

```

---

## Herramientas adicionales

Además del compilador, el paquete de desarrollo incluye algunas herramientas adicionales para cubrir todo el proceso de generar un binario y poder distribuirlo. Cada una de estas herramientas cuenta con su propio manual distribuido junto al del compilador. Todas estas herramientas pueden utilizarse por sí mismas y son totalmente independientes.

- `abasm.py` — ensamblador compatible con WinAPE y RVM. [manual](#)
  - `img.py` — conversión de imágenes a formato CPC. Puede generar pantallas de carga. [manual](#)
  - `dsk.py` — creación de disquetes .DSK. Permite distribuir los binarios generados y otros ficheros adicionales. [manual](#)
  - `cdt.py` — creación de cintas .CDT. Permite distribuir los binarios generados y otros ficheros adicionales. [manual](#)
  - `basprj` — crea una estructura básica de proyecto para comenzar a trabajar. [manual](#)
- 

## Uso del compilador

```
python abasc.py [opciones] archivo.bas [-o archivo]
```

### Opciones

- `--version` — muestra la versión del compilador.
- `-O <n>` — nivel de optimización (0 = ninguna, 1 = peephole, 2 = completa).
- `-W <n>` — nivel de las advertencias (warnings) a mostrar (0 = ninguna, 1 = solo importantes, 2 = importantes y de media importancia, 3 = todas).
- `--data <n>` — dirección de inicio para el área de datos del programa (por defecto es 0x4000, ver sección sobre Gestión de la memoria).

- `-v, --verbose` — genera archivos auxiliares del proceso de compilación (resultado del preproceso, tabla de símbolos, árbol de sintáxis, etc.).
- `-o, --out` — nombre de salida sin extensión.

## Creación de un proyecto usando BASPRJ

En ABASC, la gestión de un proyecto es sencilla. Basta con crear un fichero principal en Locomotive BASIC 2 que importe cualquier otro archivo necesario mediante el comando `CHAIN MERGE`. Tras ejecutar ABASC, se generará un fichero binario compilado. A continuación, solo será necesaria una llamada adicional a las herramientas DSK o CDT para empaquetar el resultado y poder utilizarlo en emuladores o en hardware real (por ejemplo, mediante dispositivos como Gotek, M4 o DDI-Revival).

```
python3 abasc.py main.bas
python3 dsk.py -n main.dsk --put-bin main.bin --start-addr=0x4000 --load-addr=0x0170
```

sin embargo, también es posible generar rápidamente la estructura básica de un proyecto utilizando la herramienta BASPRJ. Esta utilidad crea automáticamente un script de construcción con todo lo necesario para comenzar a trabajar: en Windows se generará un fichero `make.bat`, mientras que en Linux y macOS se creará un fichero `make.sh`. Asimismo, se incluirá un archivo `main.bas` con código de ejemplo listo para ser compilado y probado.

```
python3 basprj.py -n myproject
```

Para conocer todas las opciones disponibles, se recomienda consultar la documentación específica de BASPRJ.

## Peculiaridades del compilador

Aunque el objetivo de ABASC es permitir la compilación, sin apenas cambios, de programas escritos para BASIC 1.0 o 1.1, la propia naturaleza de un compilador —frente a un intérprete— introduce ciertas diferencias. En esta sección exploramos esos aspectos particulares que pueden tomar por sorpresa al programador acostumbrado al uso del intérprete de BASIC.

### Tipos y variables

ABASC utiliza un sistema de tipado algo más estricto que el proporcionado por el intérprete original de BASIC. Para empezar, todas las variables son de tipo **entero** por defecto, salvo que se utilice un sufijo para indicar otro tipo de dato.

Tipo	Sufijo	Notas
Entero	% (opcional)	Valores enteros en el rango -32768...32767
Real	!	Números en coma flotante de 5 bytes (4 para la mantisa y 1 para el exponente)

Tipo	Sufijo	Notas
Texto	\$	Cadenas de hasta 254 caracteres (ver siguiente sección)

## Cadenas de texto

En la implementación original de Locomotive BASIC para los Amstrad CPC, las cadenas utilizaban una estructura de **doble indirección**. Una variable de tipo texto ocupaba inicialmente 3 bytes:

- byte 1: longitud
- bytes 2 y 3: dirección al contenido de la cadena

La longitud máxima era de 255 caracteres.

En ABASC, el contenido de la cadena se almacena directamente a continuación del byte de longitud, reservando un máximo de **255 bytes para toda la estructura**, por lo que **la longitud máxima de una cadena es de 254 caracteres**.

La única excepción son las **llamadas RSX**, para las que ABASC conserva la estructura original de Locomotive BASIC con el fin de garantizar la compatibilidad. Por ello, una rutina RSX recibirá siempre las cadenas de texto en una estructura de 3 bytes:

- 1 byte: longitud
- 2 bytes: puntero al contenido

Además, es posible que el programador no desee reservar siempre los 254 bytes por defecto para una cadena, por lo que ABASC incorpora dos sentencias procedentes de Locomotive BASIC 2: **FIXED** y **DECLARE**.

Estas permiten ajustar el espacio reservado para la cadena, como en el siguiente ejemplo:

```
DECLARE A$ FIXED 10 ' La cadena A$ podrá contener hasta 10 caracteres
```

La cadena anterior reservará un total de 11 bytes (1 de longitud + 10 de contenido). Es importante destacar que, al no existir comprobaciones en tiempo de ejecución —como sí ocurre en un intérprete—, nada impide que el programador intente almacenar más caracteres de los permitidos en A\$, lo que provocará comportamientos impredecibles.

## Arrays

En Locomotive BASIC, un array que no ha sido declarado previamente con DIM se considera que tiene 10 elementos por defecto. ABASC es más estricto: la compilación fallará si el código intenta operar con arrays que no hayan sido declarados explícitamente mediante DIM.

Además, un array de cadenas reservará inmediatamente la memoria necesaria para todos sus elementos. Por defecto, cada cadena ocupa 255 bytes (1 para la longitud y 254 para el contenido), lo que puede consumir rápidamente la memoria disponible. Por ello, igual que con las cadenas individuales, es posible utilizar la cláusula **FIXED**:

```
DIM A$(5) FIXED 10 ' El espacio total será de 11 bytes × 5 elementos
```

## Estructuras con RECORD

ABASC incluye soporte para la organización de variables en estructuras más complejas denominadas **REGISTROS**. Internamente, un registro es simplemente una forma de dividir y etiquetar la memoria reservada por una cadena de texto. Para utilizar registros, el primer paso es declarar su estructura mediante la sentencia RECORD.

RECORD nombre; lista de campos

Ejemplo:

RECORD persona; nom\$ FIXED 10, edad

Los patrones definidos con RECORD pueden aplicarse a cadenas empleando el símbolo . tras el nombre de la variable:

DECLARE A\$ FIXED 13 ' No es obligatorio, pero reduce el consumo de memoria  
RECORD persona; nom\$ FIXED 10, edad ' Requiere 13 bytes de memoria

A\$.persona.nom\$ = "Juan"

A\$.persona.edad = 20

El programa anterior dejará el contenido de la memoria reservada por A\$ como sigue:

BYTE	Contenido	Valor
0 - 10	longitud y contenido de nom\$	4,J,u,a,n,0,0,0,0,0
11 - 12	valor de edad	20

## Procedimientos y Funciones

Tradicionalmente, BASIC permite organizar código reutilizable mediante rutinas invocadas con GOSUB y RETURN (sin soporte para parámetros) o mediante funciones de una sola línea definidas con DEF FN. ABASC es plenamente compatible con ambos mecanismos, pero además incorpora una forma más moderna de estructurar el código, introducida en la versión 2 Plus de Locomotive BASIC. La sintaxis es la siguiente:

SUB nombre(lista de parámetros)

....

END SUB

FUNCTION nombre(lista de parámetros)

....

END FUNCTION

Las rutinas declaradas con FUNCTION deben incluir al menos una instrucción de asignación al propio nombre de la función, que actuará como valor de retorno.

Las funciones pueden llamarse directamente como parte de una expresión, mientras que las subrutinas deben llamarse con CALL, indicando el nombre del procedimiento y los parámetros entre paréntesis separados por comas.

```

function pow2(x)
    pow2 = x * x
end function

sub message(m$)
    print m$
end sub

label MAIN
    result = pow2(2)
    msg$ = "2 * 2 is " + str$(result)
    call message(msg$)
end

```

Las variables declaradas dentro del cuerpo de un procedimiento (mediante DECLARE, DIM, incluyéndolas en la parte izquierda de una asignación o utilizándolas en INPUT, READ o LINE INPUT) son siempre locales y no pueden ser referenciadas desde el exterior. Las variables globales, por su parte, pueden emplearse dentro de un procedimiento, pero solo si aparecen al principio del cuerpo del procedimiento en una sentencia SHARED.

En cuanto a la semántica de paso de parámetros, los enteros se pasan por valor, mientras que las cadenas de texto, los números reales y los array se pasan por referencia (es decir, mediante un puntero a su contenido). Por tanto, en estos tres últimos casos es posible modificar la variable original desde el cuerpo del procedimiento. Los arrays deben pasarse en la llamada usando el sufijo [], igual que se hace en el uso del comando SHARED. En la declaración del procedimiento o función, debe indicarse el vector usando [] e indicando los índices como se hace en la declaración con DIM.

```

DIM myvec(3)

sub printvec(v[3])
    for i=0 to 3
        print v(i)
    next
end sub

myvec(0) = 0; myvec(1) = 1; myvec(2) = 2; myvec(3) = 3
call printvec(myvec[])

```

**NOTA SOBRE RECURSIVIDAD:** ABASC no permite recursividad. Al igual que ocurre con las variables globales, las variables locales reservan memoria en tiempo de compilación. Debido a ello, el código no es reentrante y no es posible realizar llamadas recursivas.

## Uso de código ensamblador

Mediante la sentencia ASM es posible incrustar código ensamblador en cualquier parte del programa BASIC. **ABASM**, el ensamblador utilizado por ABASC, dispone de su propio manual, donde se describe con detalle la sintaxis y opciones disponibles.

Además, se pueden invocar rutinas escritas en ensamblador utilizando la sentencia CALL, tal y como muestra el siguiente ejemplo:

```
ASM "mylabel: ret ; rutina vacía"
```

```
CALL "mylabel"
```

Es posible pasar argumentos a las rutinas ensambladas, aunque esto requiere conocer la convención de llamadas utilizada por ABASC. Los parámetros se apilan **en orden**, del primero al último, y la función se invoca con el registro **IX apuntando al último parámetro**. La rutina llamada **no** debe desapilar los parámetros; es el llamante quien se encarga de ello tras el retorno.

Por ejemplo, una rutina que reciba tres parámetros enteros (cada uno de 2 bytes):

```
CALL mirutina(param1, param2, param3)
```

Podrá acceder a ellos mediante el siguiente esquema:

Parámetro	Direcciones relativas
param1	IX+4, IX+5
param2	IX+2, IX+3
param3	IX+0, IX+1

Por último, es posible añadir la cláusula ASM a la declaración de una función o subrutina, indicando que todo el código va a ser ensamblador y que el compilador no necesita gestionar la memoria temporal (montículo).

```
SUB cpcSetColor(i,c) ASM
    ' Equivalent to the BASIC call INK
    ' param 1: is the ink number (0-16) bening 16 the border ink.
    ' param 2: color in hardware values - &40 (i.e &14 means &54 black)
    ASM "ld      bc,&7F00 ; Gate Array"
    ASM "ld      a,(ix+2) ; ink number"
    ASM "out     (c),a"
    ASM "ld      a,&40"
    ASM "ld      e,(ix+0) ; HW color"
    ASM "or      e"
    ASM "out     (c),a"
    ASM "ret"
END SUB
```

```
CALL cpcSetColor(0,&14)
```

Mediante ASM es posible importar a nuestro proyecto otros ficheros con código en ensamblador o ficheros binarios:

```
ASM "read 'mylib.asm'      ; codigo ensamblador adicional"
ASM "incbin 'assets.bin'  ; contenido binario"
```

## Punteros

Locomotive BASIC emplea el símbolo @ para acceder a la dirección de memoria de una variable. Por ejemplo, para leer y mostrar los 5 bytes correspondientes a un número real, se puede utilizar el siguiente código:

```

a! = 43.375
PRINT "MEMORY ADDRESS:"; @a!
PRINT "MEMORY CONTENT (HEX):"
FOR i = 0 TO 4
    PRINT i, HEX$(PEEK(@a! + i), 2)
NEXT

```

ABASC extiende el uso del símbolo @ permitiendo que se use para acceder a la dirección asociada a una etiqueta definida con LABEL, así como obtener la dirección de memoria desde la que se leerán los valores en la siguiente llamada a READ. Es, incluso, posible obtener la dirección a una etiqueta definida desde código en ensamblador. Estas opciones pueden ser muy interesantes cuando se trabaja con ficheros importados que contienen código en ensamblador o datos en binario.

```

LABEL MAIN
    CLS

    spdir = @LABEL(mysprite)
    RESTORE palette
    pldir = @DATA
    asmdir = @LABEL("asm_label")
    ' Example usage of these pointers...
END

LABEL mysprite:
    ASM "read 'my_sprite.asm'"

LABEL palette:
    DATA 1,2,3,4

ASM "asm_label:"

```

## Gestión de la memoria

El mapa de memoria de un programa compilado con ABASC es el siguiente:

Dirección	Descripción
<b>0x0040</b>	Comienzo del área para la inicialización de la aplicación y reserva de memoria temporal (montículo).
<b>_code_</b>	Comienzo del área para el código de la aplicación. Comienza justo después del código de inicialización y del montículo.
<b>_runtime_</b>	Etiqueta que marca el comienzo del área para rutinas de apoyo generadas por el compilador.
<b>_data_</b>	Etiqueta que marca el comienzo del espacio reservado para las variables. Su dirección más baja posible es 0x4000, ya que no puede compartir espacio con el área de direccionamiento del Firmware (0x0000-0x3FFF). En cualquier caso, se puede configurar mediante el parámetro <code>--data</code> . Si el código que precede a esta área ocupa la dirección designada para los datos, el compilador moverá esta zona a la primera dirección de memoria posterior que esté disponible.

Dirección	Descripción
<b>_program_end_</b>	Etiqueta que marca la dirección donde finaliza la memoria consumida por el programa.

Locomotive BASIC incluye una serie de comandos relacionados con la gestión de memoria: HIMEM, MEMORY, FRE y SYMBOL AFTER. ABASC los soporta, pero su significado varía ligeramente debido al modelo compilado:

Comando	Función ABASC
<b>HIMEM</b>	Devuelve la dirección de memoria inmediatamente posterior al final del programa.
<b>MEMORY</b>	Establece la dirección de memoria máxima a la que podrá llegar el binario generado. Si se supera, la compilación falla.
<b>SYMBOL AFTER</b>	ABASC reserva memoria para caracteres redefinibles (UDC) igual que Locomotive BASIC. Esta zona forma parte de <b>data</b> . Puede liberarse con SYMBOL AFTER 256.
<b>FRE(0)</b>	Devuelve la memoria disponible entre _program_end_ y la zona del Firmware donde empiezan las variables (&A6FC).
<b>FRE(1)</b>	Devuelve la memoria temporal (montículo) disponible en ese instante.
<b>FRE(“““)</b>	Fuerza la liberación de la memoria temporal (montículo) y devuelve el mismo valor que FRE(0).

ABASC utiliza memoria temporal para almacenar valores intermedios durante la evaluación de expresiones (por ejemplo, concatenación de cadenas o cálculo de expresiones numéricas). Esta memoria se reserva en el montículo o “heap”. Dicho montículo comienza en la zona baja de la memoria (al rededor de la dirección 0x0040) y su tamaño máximo se calcula durante la compilación. Después de cada sentencia, la memoria temporal se libera automáticamente. La única excepción ocurre durante una llamada a FUNCTION o SUB: la memoria temporal previa a la llamada se preserva para poder restaurar el contexto al regresar.

ABASC imprime al acabar de compilar un mensaje con la cantidad máxima de memoria del montículo calculada durante la compilación. Dicho valor puede usarse para ajustar el parámetro utilizado junto al flag --heap.

## Uso del Firmware

ABASC se apoya de manera extensa en las rutinas del **Firmware del Amstrad CPC**, especialmente para el manejo de números reales. Esto significa que, aunque el código compilado es más rápido que el interpretado, puede verse limitado por el rendimiento de dichas rutinas del sistema.

Sin embargo, es posible utilizar la sentencia ASM para definir alternativas más eficaces para las llamadas al Firmware (como CLS, INK, BORDER, PAPER, etc.). Sin embargo, debe tenerse en cuenta que, si no se deshabilitan las interrupciones, el Firmware seguirá activo y **podría sobrescribir los cambios realizados** sin previo aviso.

Otra opción es modificar directamente el código ensamblador del programa, ya que ABASC genera durante la compilación un fichero con extensión .ASM que contiene todo el código del programa. Esto permite al programador modificarlo o añadir optimizaciones específicas cuando sea necesario, pudiendo usar **ABASM** para obtener el binario correspondiente. Mediante la opción `--verbose` obtendremos muchos más comentarios en el fichero ASM generado, lo que nos ayudará a realizar un mejor seguimiento de la traducción de nuestras sentencias BASIC a código ensamblador.

## Librerías

La instalación de ABASC contiene un directorio llamado `lib`. Cualquier fichero .BAS puede ser dejado ahí para incluirlo desde cualquiera de nuestros programas con el comando `CHAIN MERGE`.

`CHAIN MERGE` tratará primero de resolver cualquier fichero a incluir contra el directorio local de nuestro código fuente. Si el fichero dado no es un fichero local a nuestro programa, buscará en el directorio `lib` de la instalación de ABASC al considerar que se trata de una “librería”, un fichero .BAS reusable desde cualquier proyecto. Por ejemplo, podemos probar el fichero `memory.bas` que se distribuye con ABASC mediante este simple programa:

```
CHAIN MERGE "memory.bas"
```

```
A$="Hola mundo"
B$=""
```

```
CALL MEMSET(&C000, &4000, 0)
CALL MEMCOPY(@B$, @A$, LEN(A$)+1)
PRINT B$
```

## Comandos y sintaxis del lenguaje

A continuación se muestra una breve guía de la notación y de los comandos y funciones soportadas. No pretende ser una guía exhaustiva sobre Locomotive BASIC, sino recoger aquellos aspectos particulares del compilador. Para aprender más sobre el lenguaje, se recomienda consultar las obras listadas en la sección Referencias, al principio de este manual.

## Notación

Caracteres especiales:

carácter	Notas
& o &H	prefijo para números en hexadecimal
&X	prefijo para números en binario

carácter	Notas
:	separador de sentencias en la misma línea
#	prefijo para denotar un canal de texto (0-9)
”	delimitador de cadenas de texto
@	delante del nombre de una variable indica dirección de memoria apuntada por dicha variable
	delante de un identificador indica llamada a función RSX

## Listado de comandos y funciones

### ABS(<numeric expression>)

Función. Devuelve el valor absoluto del número proporcionado como parámetro. La expresión numérica puede ser entera o real.

### AFTER delay[,timer] GOSUB etiqueta

Comando. Llama a una subrutina indicada tras un retardo. El “delay” se mide con un grano de 1/50 segundos. El segundo parámetro (opcional) indica cuál de los cuatro temporizadores se debe utilizar (0..3). Si no se especifica, se utiliza el valor 0 por defecto. Como etiqueta para la sentencia GOSUB se puede usar tanto un número de línea (INT) como un literal definido por la sentencia LABEL.

ABASC emplea las funciones del Firmware para la gestión de eventos asíncronos. Las rutinas del usuario son llamadas con la ROM baja activa y, por tanto, el código debería mantenerse breve y no hacer uso de los primeros 16K de memoria. Por ejemplo, las operaciones con números en coma flotante o las operaciones con textos tratarán de reservar memoria temporal en dicho rango y deberían evitarse. Las operaciones con enteros, en cambio, no deberían dar problemas. Este mecanismo también depende de que las interrupciones estén activas (ver DI y EI).

```
A = 0
AFTER 50 GOSUB INCR ' Llama a la rutina INCR después de 1 segundo
A = 5
END

LABEL INCR
    PRINT A
RETURN
```

### ASC(string)

Función. Devuelve el valor ASCII del primer carácter de la cadena suministrada como parámetro.

```
PRINT ASC("HOLA") ' imprime 72, el código ASCII para la letra H
```

**ASM string[,string]\***

Comando. Inserta el código contenido de la lista de cadenas de texto como código ensamblador. Cada cadena de la lista se inserta como una nueva línea.

```
ASM "ld hl,_my_str", "ld a,(hl)"
```

**ATN(x)**

Función. Devuelve la arcotangente (arctan) de x. Implica el uso de números reales.

**AUTO linenumber[,increment]**

Comando. ABASC ignora este comando y emite un mensaje de alerta sobre su uso, ya que no tiene utilidad para un programa compilado.

**BIN\$(number,digits)**

Función. Devuelve el valor entero number como cadena de texto con su representación binaria. Locomotive BASIC permite especificar el número exacto de dígitos a utilizar en la representación binaria, pero **ABASC solo soporta los valores 8 o 16**.

```
PRINT BIN$(16,8) ' imprimira la cadena de texto "00010000"
```

**BORDER colour1[,colour2]**

Comando. Permite especificar el color del borde. Si se proporcionan dos valores, se produce un parpadeo cuyo tiempo controla el comando SPEED INK.

```
BORDER 0,1
```

**CALL address[,list of parameters]**

Comando. Permite llamar a una rutina existente en memoria indicando su dirección, a una rutina declarada con SUB o FUNCTION, o a una etiqueta declarada dentro de un bloque en ensamblador.

```
SUB nada
    print "solo imprimo nada"
END SUB
```

```
CALL &BC14 ' rutina del firmware para limpiar la pantalla
CALL nada()
CALL "bucle_eterno"
PRINT "aqui no llegaremos"
ASM "bucle_eterno: jr bucle_eterno"
```

## CAT

Comando. Muestra el contenido del dispositivo de almacenamiento actual. Es posible cambiar el dispositivo a través de llamadas a funciones RSX como |TAPE, |DISC, |A o |B.

## CHAIN

Comando. En BASIC, se utiliza para reemplazar el programa actual en memoria por otro. ABASC ignora esta instrucción y emite una advertencia si la encuentra en el código.

## CHAIN MERGE string

Comando. ABASC reinterpreta este comando para permitir dividir nuestro código entre varios ficheros. `string` debe ser una ruta a un fichero .BAS alcanzable desde el fichero donde se hace la referencia.

Si `string` no es un fichero local al programa, buscará en el directorio `lib` de la instalación de ABASC al considerar que se trata de una “librería”, un fichero .BAS reusable desde cualquier proyecto.

```
fichero OTRO.BAS
    MYVAR$ = "UNA CADENA MUY UTIL"
```

```
fichero MAIN.BAS
    CHAIN MERGE "OTRO.BAS"
    PRINT MYVAR$
END
```

## CHR\$(x)

Función. Devuelve una cadena de texto con el carácter equivalente indicado por `x` en el rango 0-255.

```
PRINT CHR$(250)
```

## CINT(x)

Función. Devuelve un entero con la conversión redondeada del número real `x`. `x` debe estar dentro del rango -32768..32767 o el valor devuelto será erróneo.

```
PRINT CINT(PI)
```

## CLEAR

Comando. Fija todas las variables numéricas a 0 y los strings a "", cierra cualquier fichero abierto y vuelve a poner el modo para ángulos a RAD.

## **CLEAR INPUT**

Comando. Este comando se introdujo con la versión BASIC 1.1. ABASC permite su uso incluso en un Amstrad CPC 464 utilizando la rutina del Firmware KM RESET en vez de KM FLUSH.

## **CLG [tinta]**

Comando. Borra la pantalla de gráficos usando el valor actual de PAPER. Si tinta está presente, se fija como nuevo valor para PAPER antes del borrado.

## **CLOSEIN**

Comando. Cierra el fichero abierto actualmente para lectura. Ver OPENIN.

## **CLOSEOUT**

Comando. Cierra el fichero abierto actualmente para escritura. Ver OPENOUT.

## **CLS [#x]**

Comando. Borra la pantalla usando el color de PAPER actual. Es posible indicar un canal con #x. Los valores 0-7 están disponibles para definir áreas de la pantalla mediante el comando WINDOW, mientras que el valor #8 suele estar asociado a la impresora (no soportado por ABASC) y el #9 se asocia con ficheros.

## **CONST**

Comando. CONST define una constante numérica simple y le asigna un nombre como si fuera una variable más. A partir de ese momento, cuando el nombre de la constante aparezca como parte de una expresión, será sustituido directamente por el valor numérico, lo que puede habilitar ciertas optimizaciones del compilador. Además, si se intenta cambiar su valor inicial, la compilación fallará alertando al programador del error.

```
CONST VMEM = &C000
```

```
FOR I=0 TO 16384  
    POKE VMEM + I, &FF  
NEXT
```

## **CONT**

Comando. En el BASIC original permite continuar la ejecución de un programa detenido por las instrucciones BREAK, STOP o END. En un programa compilado no tiene sentido y ABASC lo redefine para detener el programa y esperar la pulsación de cualquier tecla antes de continuar, lo que puede ser útil para depurar programas.

**COPYCHR\$(#canal)**

Función. Devuelve el carácter situado en la posición actual del cursor de texto para el canal indicado. Esta función apareció con la versión BASIC 1.1. ABASC proporciona una implementación que permite utilizar la función incluso en programas que se van a ejecutar en un Amstrad CPC 464.

```
MODE 1
PRINT "HELLO WORLD"
LOCATE 3,1
C$ = COPYCHR$(#0) ' la letra L
LOCATE 1,2: PRINT C$
```

**COS(x)**

Función. Devuelve el coseno de x. Implica el uso de números reales.

**CREAL(x)**

Función. Convierte x (normalmente un número entero) en un número real.

**CURSOR sistema[,usuario]**

Comando. Incorporado en la versión 1.1 de BASIC. Permite fijar el valor encendido 1 o apagado 0 a los flags de visibilidad del cursor. El cursor solo se mostrará cuando ambos valores (sistema y usuario) estén a 1.

**DATA lista-de-constantes**

Comando. Permite añadir al programa una serie de valores (números enteros o caracteres) que después pueden leerse en orden mediante la instrucción READ.

```
CLS
FOR I=0 TO 5
    READ nom$
    PRINT "Nombre:", nom$
NEXT
END
```

```
DATA "Xavier","Ross","Gada",
DATA "Anabel","Rachel","Elvira"
```

**DECLARE variable[\$ FIXED longitud]),...**

Comando. Este comando apareció con la versión 2 de Locomotive BASIC. Permite “dar a conocer” una variable que se va a utilizar posteriormente. Normalmente, solo es necesario declarar los arrays mediante DIM ya que las variables quedan declaradas en cuanto se les asigna un valor. Sin embargo, DECLARE permite crear cadenas de texto con una longitud máxima menor a los 254 bytes utilizados por defecto o declarar variables enteras con el valor por defecto de 0 reduciendo el código e instrucciones generadas, ya que no es necesaria una asignación inicial.

por ejemplo:

```
B$ = ""           ' B$ reserva espacio para 254 caracteres
DECLARE A$ FIXED 15 ' A$ reserva espacio para 15 caracteres
B = 0             ' B queda inicializada a 0 generando más código
ensamblador
DECLARE A         ' que A.
```

## DEC\$(numero,patron)

Función. Esta función apareció con la versión 1.1 de BASIC. Permite convertir numero en una cadena aplicando un patrón para indicar el número de espacios antes o después del punto decimal. ABASC no soporta todavía el uso de estos patrones, por lo que esta llamada se comporta, básicamente, igual que STR\$.

```
PRINT DEC$(15.5, "###.##")
```

## DEF FN nombre(parametros)=expresion

Comando. Permite declarar una función que aplicará la expresión de la derecha a los valores indicados como parámetros en cada llamada. En BASIC 1.0 era la única forma de declarar funciones. ABASC soporta FUNCTION ... END FUNCTION que es un mecanismo mucho más versátil.

A diferencia de Locomotive BASIC **las funciones y procedimientos DEBEN declararse antes de su uso**. También es importante resaltar que debido a la gestión de tipos más estricta, una función que devuelva un valor real debe terminar su nombre obligatoriamente con el sufijo !, igual que una que devuelva una cadena de texto debe hacerlo con \$.

```
DEF FNinteres!(principal)=principal * 1.14
PRINT FNinteres!(1000)
```

## DEFINT, DEFSTR, DEFREAL

Comandos. Originalmente fijaban un rango de letras iniciales para indicar que una variable era de un tipo determinado. Como ABASC utiliza un sistema más restrictivo de tipos, estos comandos no tienen ningún efecto. El programador debe emplear obligatoriamente los sufijos %, ! y \$ para indicar el tipo de una variable.

## DEG

Comando. Establece que las funciones que trabajan con ángulos devuelvan sus resultados en grados en vez de radianes.

```
DEG
PRINT SIN(90.0)
RAD
PRINT SIN(90.0)
```

## **DELETE bajo-alto**

Comando. En Locomotive BASIC este comando borraba un conjunto de líneas del programa en BASIC. En ABASC, este funcionamiento no tiene sentido, así que DELETE se ha modificado para permitir borrar (llenar con 0s) una región de la memoria. El rango debe proporcionarse como: dirección inicial - dirección final.

```
DELETE &C000-&FFFF
```

## **DERR**

Comando. Introducido en la versión 1.1 de BASIC. Almacenaba el último error producido al trabajar con la unidad de disco. ABASC ignora cualquier referencia a este comando y emite una advertencia al respecto si lo encuentra en el código.

## **DI**

Comando. Desactiva el mecanismo de interrupciones. Con las interrupciones desactivadas, dejará de actualizarse el valor devuelto por TIME y la gestión de eventos registrados con AFTER o EVERY. Las interrupciones pueden volverse a activar con el comando EI.

## **DIM array(indice1, indice2, ...) [FIXED longitud]**

Comando. Permite declarar y reservar la memoria a utilizar por un array (vector). El tipo de dato debe indicarse como sufijo al nombre del array (% , ! , \$). Si no se indica ningún sufijo, los datos serán enteros. En el caso de un array de cadenas de texto, es posible reducir el tamaño máximo reservado para cada cadena usando la cláusula FIXED después de la lista de índices.

Los índices van desde 0 hasta el número indicado en la declaración.

```
DIM nom$(3) FIXED 8
```

```
nom$(0) = "Juan"
nom$(1) = "Daniel"
nom$(2) = "Pepe"
nom$(3) = "Roberto"
```

```
FOR I=0 TO 3
    PRINT nom$(I)
NEXT
```

## **DRAW x,y[,i[,modo]]**

Comando. Dibuja una línea desde la posición actual del cursor hasta la posición x e y. Si se indica un tercer parámetro, este es el color a utilizar. Con la versión 1.1 de BASIC, se añadió un cuarto parámetro soportado por ABASC (incluso para programas que correrán en un Amstrad CPC 464). Este cuarto parámetro indica el modo o máscara a aplicar entre cada punto de la línea y el fondo, con los valores que se indican a continuación:

Valor	Modo
0	Fill (normal)
1	XOR (OR eXclusivo)
2	AND
3	OR

MODE 1

DRAW 100,100,1

DRAW 0,100,2

DRAW 100,0,3

DRAW 0,0,2

### **DRAWR x,y[,i[,modo]]**

Comando. Al igual que DRAW dibuja una línea, aunque los valores de x e y no son posiciones absolutas de la pantalla sino valores relativos a la posición actual. El resto de parámetros tienen el mismo significado que en la instrucción DRAW.

### **EDIT línea[-línea]**

Comando. En Locomotive BASIC permite editar una línea de código. En ABASC este comando no tiene sentido y es ignorado si forma parte del código a compilar.

### **EI**

Comando. Activa las interrupciones. Ver DI.

### **END**

Comando. Termina la ejecución del programa. Mientras que en el interprete de BASIC esto significa devolver el control al usuario, ABASC salta a un bucle infinito. STOP, en cambio, fuerza un reinicio de la máquina.

### **END FUNCTION**

Comando. Termina la declaración de una función. Ver FUNCTION.

### **END SUB**

Comando. Termina la declaración de un procedimiento. Ver SUB.

### **ENT numero de envolvente, secciones**

Comando. Define la variación en tono de un sonido. Locomotive BASIC permite especificar dos tipos de envolventes de tono (secciones), una con tres parámetros y otra con dos. Aunque no está documentado, para diferenciarlas, es posible utilizar el símbolo = antes del primer número en el segundo caso. ABASC no falla si se encuentra dicho carácter, pero utiliza el número de parámetros

para saber si nos encontramos en el primer caso o en el segundo. En caso de duda, procederá siempre considerando que estamos usando el primer caso, donde cada envolvente se especifica usando tres valores.

Sección tipo 1: \* Parámetro 1: número de escalores, de 0 a 239. \* Parámetro 2: tamaño de cada escalón, de -128 a +127. \* Parámetro 3: pausa

Sección tipo 2: \* Parámetro 1: periodo del tono (entero de 16 bits). \* Parámetro 2: pausa

## ENV número de envolvente, secciones

Comando. Define la variación en volumen de un sonido. Locomotive BASIC permite especificar dos tipos de envolventes de volumen (secciones), una con tres parámetros y otra con dos. Aunque no está documentado, para diferenciarlas, es posible utilizar el símbolo = antes del primer número en el segundo caso. ABASC no falla si se encuentra dicho carácter, pero utiliza el número de parámetros para saber si nos encontramos en el primer caso o en el segundo. En caso de duda, procederá siempre considerando que estamos usando el primer caso, donde cada envolvente se especifica usando tres valores.

Sección tipo 1: \* Parámetro 1: número de escalores, de 0 a 127. \* Parámetro 2: tamaño de cada escalón, de -128 a +127. \* Parámetro 3: pausa, rango de 0 a 255.

Sección tipo 2: \* Parámetro 1: ID de envolvente según el hardware de sonido. \* Parámetro 2: periodo de la envolvente. Valor que se manda a los registros.

ENV 1,=9,2000

ENV 2,127,0,0,127,0,0,127,0,0,127,0,0,127,0,0

ENV 3,=9,9000

## EOF

Función. Permite saber si el fichero del que se está leyendo ha llegado al final. Devuelve -1 (true) si el final se ha alcanzado o 0 (false) en cualquier otro caso.

```
OPENIN "DATOS.TXT"
WHILE NOT EOF
    LINE INPUT #9,C$
    PRINT C$
WEND
CLOSEIN
```

## ERASE arrayname

Comando. En Locomotive BASIC permite liberar la memoria reservada por un array. ABASC reserva el espacio durante la compilación, por lo que este comando no tiene utilidad y es ignorado si forma parte del código compilado.

## ERL

Comando. En Locomotive BASIC permite conocer la línea donde se ha producido el último error. En un programa compilado no tiene utilidad y es ignorado si forma parte del código compilado.

## ERR

Comando. Permite recuperar un código de error (entero) que se haya establecido antes llamando al comando ERROR. También puede almacenar el código de error 31 (File not open) si los comandos OPENIN u OPENOUT fallan.

```
ERROR 5  
PRINT ERR
```

## ERROR integer

Comando. ABASC permite utilizar este comando para fijar un número de error que puede consultarse después con ERR.

## EVERY tiempo[,temporizador] GOSUB etiqueta

Comando. Fija el temporizador indicado (0..3 - 0 por defecto) para saltar a etiqueta cada intervalo de tiempo. El tiempo tiene un grano de 1/50 segundos, por lo que un valor de 50 quiere decir llamar a la etiqueta cada segundo.

ABASC emplea las funciones del Firmware para la gestión de eventos asíncronos. Las rutinas del usuario son llamadas con la ROM baja activa y, por tanto, el código debería mantenerse breve y no hacer uso de los primeros 16K de memoria. Por ejemplo, las operaciones con números en coma flotante o las operaciones con textos tratarán de reservar memoria temporal en dicho rango y deberían evitarse. Las operaciones con enteros, en cambio, no deberían dar problemas. Este mecanismo también depende de que las interrupciones estén activas (ver DI y EI).

```
A=0  
EVERY 300 GOSUB INCA ' imprime e incrementa A cada 6 segundos  
END
```

```
LABEL INCA  
    PRINT A  
    A=A+1  
    RETURN
```

## EXIT FOR

Comando. Aunque en Locomotive BASIC era posible abandonar un bucle con una sentencia GOTO, en un programa compilado con ABASC se producirán errores inesperados durante la ejecución. La forma adecuada de abandonar un bucle FOR antes de su terminación es utilizando el comando EXIT FOR, que saltará a la siguiente instrucción tras el NEXT. Este comando fue introducido con la versión 2 de Locomotive BASIC.

```
FOR I=0 TO 100  
    IF I = 50 THEN EXIT FOR  
NEXT  
PRINT I
```

## EXIT WHILE

Comando. Aunque en Locomotive BASIC era posible abandonar un bucle con una sentencia GOTO, en un programa compilado con ABASC se producirán errores inesperados durante la ejecución. La forma adecuada de abandonar un bucle WHILE antes de su terminación es utilizando el comando EXIT WHILE, que saltará a la siguiente instrucción tras el WEND. Este comando fue introducido con la versión 2 de Locomotive BASIC.

```
I=0
WHILE I < 101
  IF I=50 THEN EXIT WHILE
  I=I+1
WEND
PRINT I
```

## EXP(x)

Función. Calcula E elevado a x, siendo E 2.7182818 aproximadamente, el número cuyo logaritmo natural es 1. Implica el uso de números reales.

## FILL

Comando. Solo disponible para ordenadores Amstrad CPC 664, 6128 o superiores. Rellena un área de la pantalla a partir de la posición actual del cursor gráfico con la tinta activa. Permite rellenar figuras, y aunque ABASC compile el programa adecuadamente, fallará si trata de ejecutarse en un Amstrad CPC 464.

```
MODE 0
GRAPHICS PEN 15
MOVE 200,0
DRAW 200,400
MOVE 639,0
FILL 15
```

## FIX(x)

Función. Convierte a entero el número real x truncándolo. Para que el valor devuelto sea correcto, el número real debe estar en el rango -32768 to +32767.

```
PRINT FIX(PI+0.5), CINT(PI+0.5)
```

## FOR variable=inicio TO fin STEP variacion

Comando. Permite especificar un bucle donde variable variará de valor desde inicio a fin. Si no se especifica una variacion, el incremento será de 1 en cada pasada del bucle.

```
CLS
T! = TIME
FOR i=1 to 10
  FOR j=1 to 1000
```

```

        s = 1000 + j
    NEXT j
    PRINT ".";
NEXT i
PRINT " FIN!"
PRINT TIME-T!

```

## FRAME

Comando. Hace que el programa se detenga hasta la siguiente señal de sincronismo vertical del monitor (50 veces por segundo como máximo).

## FRE(x)

Función. Según el valor de x permite obtener varios valores relacionados con la memoria:

Valor del parámetro | Valor devuelto |

**FRE(0)** | Devuelve la memoria disponible entre `_program_end_` y la zona del Firmware donde empiezan las variables (&A6FC). | **FRE(1)** | Devuelve la memoria temporal disponible en ese instante. | **FRE(“““)** | Fuerza la liberación de la memoria temporal (montículo) y devuelve el mismo valor que `FRE(0)`. |

## FUNCTION nombre(parametros) [ASM]

Comando. Introducido con la versión 2 plus de Locomotive BASIC, este comando permite declarar funciones de forma parecida a `DEF FN` pero cuyo cuerpo se extienda por más de una línea.

Las rutinas declaradas con `FUNCTION` deben incluir al menos una instrucción de asignación al propio nombre de la función, que actuará como valor de retorno. Las funciones pueden llamarse directamente como parte de una expresión.

```

function pow2(x)
    pow2 = x * x
end function

```

```
result = pow2(2)
```

La cláusula `ASM` en la declaración de la función permite indicar que todo el cuerpo de la función será código en ensamblador (a través del comando `ASM`), tal y como se explica en la sección `Uso de código ensamblador` del capítulo `Peculiaridades del compilador`.

Se recomienda al programador leer la sección `Procedimientos y Funciones` para obtener más información sobre el tratamiento de los parámetros o el soporte a la recursividad en el capítulo `Peculiaridades del compilador`.

## GOSUB etiqueta

Comando. Salta a una etiqueta definida como un número de línea o como un literal decalrado con `LABEL`. Vuelve a la línea posterior al `GOSUB` al encontrar la sentencia `RETURN`.

```
A=0
GOSUB incrementar
GOSUB incrementar
PRINT A
END
```

```
LABEL incrementar
    A=A+1
RETURN
```

## **GOTO etiqueta**

Comando. Salta a una etiqueta definida como un número de línea o como un literal decalrado con LABEL.

## **GRAPHICS PAPER tinta**

Comando. Establece el valor de tinta (0..15) a utilizar como color de fondo para los caracteres escritos si se ha utilizado previamente la sentencia TAG. También como color al borrar la ventana mediante las llamadas a CLG.

```
MODE 0
MASK 15
GRAPHICS PAPER 3
DRAW 640,0
```

## **GRAPHICS PEN tinta,modo**

Comando. Introducido en la versión 1.1 de BASIC. Establece el valor de tinta (un valor entre 0 y 15) como color para las instrucciones de dibujo de líneas y puntos. El modo se refiere a como debe combinarse el dibujo con el fondo.

- 0: Fondo opaco.
- 1: Fondo transparente.

El valor de fondo solo puede utilizarse si el programa va a ejecutarse sobre máquinas CPC 664 o superiores, ya que en el Amstrad CPC 464 no está soportado y su uso acarreará efectos indefinidos.

```
MODE 0
GRAPHICS PEN 15
MOVE 200,0
DRAW 200,400
MOVE 639,0
FILL 15
```

## **HEX\$(x,digitos)**

Función. Devuelve una cadena de texto con la conversión de x a número hexadecimal. Locomotive BASIC permite especificar cualquier número de dígitos. ABASC solo soporta 2 o 4.

```
PRINT HEX$(255,2)
PRINT HEX$(2048,4)
```

## HIMEM

Función. Devuelve la dirección de memoria inmediatamente posterior al final del programa compilado por ABASC. Puede ser muy útil con el comando LOAD para cargar otros binarios en una zona libre de la memoria.

```
PRINT "Limite de la memoria consumida", HIMEM
PRINT "Memoria libre antes de las variables del Firmware", FRE(0)
```

## IF expression THEN expression ELSE expression END IF

Comando. ABASC soporta la estructura tradicional de IF .. THEN .. ELSE en una línea de Locomotive BASIC 1.0 y 1.1. Además, añade soporte para la sintaxis introducida en Locomotive BASIC 2 Plus, que permite definir el cuerpo de las sentencias THEN y ELSE en varias líneas. No es posible combinar los dos variantes en una misma sentencia IF, si se usa la forma multilinea en el cuerpo del THEN, también debe hacerse así en el cuerpo del ELSE (si está presente) y terminar la sentencia con END IF.

```
PAS$="Por favor"
LABEL PREGUNTA
  PRINT "DAME LA CONTRASEÑA:";
  INPUT C$
IF C$=PAS$ THEN
  PRINT "ADELANTE!"
ELSE
  PRINT "PRUEBA OTRA VEZ"
  GOTO PREGUNTA
END IF
END
```

## INK tinta,color1[,color2]

Comando. Asigna color1 a tinta. Si se da un segundo color, la tinta parpadeara entre color1 y color2. En número de tintas disponibles varía con el modo de la pantalla:

- Modo 2: 2 tintas (0 y 1)
- Modo 1: 4 tintas (0..3)
- Modo 0: 16 tintas (0..15)

El rango de colores va de 0 (negro) al 26 (blanco brillante).

```
MODE 1
BORDER 0
INK 0,0: INK 1,26: INK 2,26,0
PRINT "READY"
PEN 2: PRINT "_"
```

**INKEY(tecla)**

Función. Esta función analiza el teclado para determinar qué teclas se están pulsando. El escaneo se realiza 50 veces por segundo. Las teclas [MAYÚS] y [CTRL] se identifican de la siguiente manera:

Valor devuelto	[MAYÚS]	[CTRL]	Tecla especificada
-1	N/A	N/A	Sin pulsar
0	Sin pulsar	Sin pulsar	Pulsada
32	Pulsada	Sin pulsar	Pulsada
128	Sin pulsar	Pulsada	Pulsada
160	Pulsada	Pulsada	Pulsada

CLS

LABEL BUCLE

IF INKEY(55)=32 THEN PRINT "V + mayusculas": END

GOTO BUCLE

**INKEY\$**

Función. Devuelve una cadena de texto con la tecla pulsada. Si no hay ninguna tecla pulsada, devuelve la cadena vacía "".

MODE 1

LABEL BUCLE

K\$ = INKEY\$

if K\$ <> "" THEN PRINT K\$;

GOTO BUCLE

**INP(puerto)**

Función. Lee un valor del puerto de Entrada/Salida indicado.

**INPUT [#canal,]["mensaje"][;]variable1,variable2...**

Comando. INPUT es un comando muy versatil con muchas opciones. Por ello, queda fuera de este manual y se aconseja al lector consultar cualquiera de las obras listadas en el capítulo sobre Referencias.

**INSTR([posición,]cadena1,cadena2)**

Función. Busca en cadena1 la primera aparición cadena2. Si se indica el parámetro opcional posición, la búsqueda comenzará desde esa posición; de lo contrario, la búsqueda comienza desde el primer carácter. Las posiciones comienzan en 1 y no en 0.

POSA = INSTR(1,"AMSTRAD", "A")

PRINT POSA

POSA = INSTR(POSA+1, "AMSTRAD", "A")

PRINT POSA

```
POSA = INSTR(POSA+1, "AMSTRAD", "A")
PRINT POSA
```

## INT(x)

Función. Con números positivos se comporta como FIX. Con números negativos devuelve un negativo superior a FIX.

## JOY(joystick)

Función. Similar a INKEY, pero para joysticks. El valor joystick debe ser 0 o 1, pues los Amstrad CPC solo soportan dos joysticks simultáneos. Si no hay ninguna dirección o botón del joystick en uso, devuelve 0. En cualquier otro caso, devuelve un entero que codifica el estado como sigue:

Bit	Decimal	Función
0	1	Arriba
1	2	Abajo
2	4	Izquierda
3	8	Derecha
4	16	Fuego 2
5	32	Fuego 1

## KEY tecla,cadena

Comando. Asocia una cadena de texto a una tecla de función. ABASC no soporta este comando y emite un warning si lo encuentra como parte del código a compilar.

## KEY DEF tecla,repeticion[,<normal>[,<mayus>[,<control>]]]

Comando. Redefine que devuelve la pulsación de tecla. ABASC no soporta este comando y emite un warning si lo encuentra como parte del código a compilar.

## LABEL etiqueta

Comando. Define una etiqueta a la que se puede saltar con GOTO o GOSUB. etiqueta es un identificador y no una cadena de texto, por lo que no debe enmarcarse entre dobles comillas. Las etiquetas no tienen en cuenta la diferencia entre mayúsculas o minúsculas.

```
LABEL main
    PRINT "HOLA MUNDO"
GOTO MAIN
```

Junto al caracter @ puede utilizarse para obtener la dirección en memoria de una etiqueta (definida en BASIC o ensamblador) o la dirección desde la que se leerán datos en la próxima llamada a READ.

```
LABEL MAIN
CLS
```

```

    spdir = @LABEL(mysprite)
    RESTORE palette
    pldir = @DATA
    asmdir = @LABEL("asm_label")
    ' Example usage of these pointers...
END

```

```

LABEL mysprite:
    ASM "read 'my_sprite.asm'"

```

```

LABEL palette:
    DATA 1,2,3,4

```

```

ASM "asm_label:"

```

### **LEFT\$(cadena,n)**

Función. Devuelve los primeros n caracteres de la izquierda de cadena.

```

PRINT LEFT$("AMSTRAD", 3)

```

### **LEN(cadena)**

Función. Devuelve la longitud en caracteres de cadena

```

PRINT LEN("AMSTRAD")

```

### **LET variable=expression**

Comando. Un vestigio de las primeras especificaciones de BASIC. No es necesario emplear este comando para realizar asignaciones en Locomotive BASIC, aunque se soporta su uso por compatibilidad.

### **LINE INPUT [#canal,][;][cadena;]<variable>**

Comando. Acepta una línea de texto del canal indicado (#0 por defecto). El canal #9 se utiliza para leer del fichero de entrada abierto. Para el resto de canales #0-#8 se comporta, practicamente, como el comando INPUT.

```

OPENIN "DATOS.TXT"
WHILE NOT EOF
    LINE INPUT #9,C$
    PRINT C$
WEND
CLOSEIN

```

### **LIST [rango de líneas][,#canal]**

Comando. ABASC ignora este comando y emite una advertencia si lo encuentra como parte del código a compilar.

**LOAD fichero[,dirección]**

Comando. Carga un fichero de disco o cuenta en memoria. ABASC solo soporta la carga de binarios. Si se proporciona el segundo parámetro, cargará el programa en la dirección indicada.

DIREC = HIMEM

LOAD "SPRITES.BIN",DIREC

**LOCATE [#canal,]x,y**

Comando. Posiciona el cursor de texto en la posición x e y. Las posiciones empiezan en 1 y el tamaño en x depende del modo gráfico (0 = 20, 1 = 40, 2 = 80). Si se indica un #canal los límites dependerán de las dimensiones especificadas con WINDOW.

CLS

LABEL MAIN

FRAME

FOR x=2 TO 39:

LOCATE x-1,10: PRINT " "

LOCATE x,10: PRINT CHR\$(250)

NEXT

GOTO MAIN

**LOG(x)**

Función. Devuelve el logaritmo natural de x. Implica el uso de números reales.

**LOG10(x)**

Función. Devuelve el logaritmo en base 10 de x. Implica el uso de números reales.

**LOWER\$(cadena)**

Función. Devuelve cadena con todos sus caracteres pasados a minúsculas.

C\$="AmSTRaD"

PRINT LOWER\$(C\$)

PRINT UPPER\$(C\$)

**MASK mascara[,puntoinicial]**

Comando. Solo disponible a partir de BASIC 1.1. Cualquier programa compilado usando este comando solo funcionará en ordenadores Amstrad CPC664 y CPC6128. Establece la máscara o plantilla que se utilizará al dibujar líneas. El valor binario mascara debe estar en el rango de 0 a 255. Su significado es activar (1) o desactivar (0) los bits de cada grupo adyacente de 8 píxeles. puntoinicial determina si el primer punto de la línea se debe dibujar (1) o no (0).

MODE 0

MASK 15 ' mascara = 00001111

GRAPHICS PAPER 3  
DRAW 640,0

## **MAX(a,b[,c,d,e...])**

Función. Devuelve el máximo valor de entre los proporcionados como parámetros. ABASC soporta tanto el uso de números enteros como reales.

## **MEMORY maxdir**

Comando. Establece maxdir como la dirección máxima en memoria que puede llegar a ocupar el binario generado por el programa compilado. Si se supera, la compilación falla.

MEMORY &A6FB ' En &A6FC comienzan las variables del Firmware/AMSDOS

## **MERGE fichero**

Comando. Lee fichero de disco o cinta y reemplaza el programa en memoria. ABASC no soporta este comando y producirá un error si lo encuentra como parte del código del programa. Para añadir otros binarios (o reemplazarlos tras su uso) se recomienda utilizar el comando LOAD.

## **MID\$(cadena,inicio[,n])**

Función y Comando. Como función en una expresión, devuelve el número de caracteres n desde la posición inicio. Como comando, puede utilizarse para reemplazar una parte de la cadena. La escritura en memoria siempre es delicada y el programador debe tener cuidado de no sobrepasar los límites de almacenamiento de la cadena o el programa se comportará de manera inesperada.

```
C$="AMSTRAD"
PRINT MID$(C$,3,3)
MID$(C$,3,3) = "BBB"
PRINT C$
```

## **MIN(a,b[,c,d,e,f...])**

Función. Devuelve el mínimo valor de entre los proporcionados como parámetros. ABASC soporta tanto el uso de números enteros como reales.

## **MODE n**

Comando. Cambia el modo de pantalla a 0, 1 o 2.

## **MOVE x,y[tinta[,modo]]**

Comando. Mueve el cursor gráfico a la posición x e y. Si proporciona un tercer parámetro, este indica la tinta con el color a usar a partir de ese momento. El cuarto parámetro indica el modo o máscara a aplicar entre cada punto de las líneas y el fondo, con los valores que se indican a continuación:

Valor	Modo
0	Fill (normal)
1	XOR (OR eXclusivo)
2	AND
3	OR

## **MOVER x,y[,tinta[,modo]]**

Comando. Igual que MOVE, pero siendo x e y valores relativos a la posición actual en vez de posiciones absolutas.

## **NEW**

Comando. En Locomotive BASIC borra el programa actual y sus variables de la memoria. ABASC emite código para reiniciar la máquina (CALL 0).

## **NEXT variable**

Comando. Delimita un bucle FOR.

## **ON n GOSUB lista de etiquetas**

Comando. Salta a la etiqueta de la lista indicada por n y regresa tras encontrar un RETURN. Las etiquetas empiezan en 1. Pueden ser números de línea o identificadores declarados con LABEL.

## **ON n GOTO lista de etiquetas**

Comando. Salta a la etiqueta de la lista indicada por n. Las etiquetas empiezan en 1. Pueden ser números de línea o identificadores declarados con LABEL.

## **ON BREAK GOSUB etiqueta**

Comando. Este comando salta a etiqueta cuando un programa se interrumpe por una pulsación doble de la tecla ESC. Los programas compilados por ABASC no pueden detenerse de esta manera por lo que este comando es ignorado y se emite una alerta si aparece como parte del código a compilar.

## **ON BREAK STOP**

Comando. Desactiva la última sentencia ON BREAK GOSUB que se haya emitido. Como Los programas compilados por ABASC ignoran la sentencia anterior, este comando también es ignorado y se emite una alerta si aparece como parte del código a compilar.

## ON ERROR GOTO etiqueta

Comando. Este comando salta a etiqueta cuando se detecta un error en un programa BASIC durante su ejecución. ABASC solo salta a etiqueta si el valor de ERR es diferente de 0, por ejemplo, porque se ha fijado otro valor mediante el comando ERROR.

**NOTE:** Hay que tener cuidado de no salir de un bucle WHILE o FOR usando este comando.

```
ERROR 0
ON ERROR GOTO errormsg
ERROR 1
ON ERROR GOTO errormsg
PRINT "Sin errores"
END
```

```
LABEL errormsg
    print "Error", ERR
END
```

## ON SQ (canal) GOSUB etiqueta

Comando. Registra el salto a una etiqueta como una interrupción que debe ejecutarse cuando hay un “slot” libre en la cola de sonido indicada por canal. El valor de canal debe ser un valor de entre los de la siguiente lista: \* 1 = canal A \* 2 = canal B \* 4 = canal C

```
ON SQ(2) GOSUB INSERTAenB
```

## OPENIN fichero

Comando. Abre el fichero indicado por fichero para lectura. Se puede consultar un ejemplo en el apartado sobre la función EOF. En caso de error produce el código 31 que puede ser consultado con ERR. Solo un fichero puede estar abierto a la vez para lectura.

## OPENOUT fichero

Comando. Abre el fichero indicado por fichero para escritura. En caso de error produce el código 31 que puede ser consultado con ERR. Solo un fichero puede estar abierto a la vez para escritura.

## ORIGIN x,y[,izq,der,arriba,abajo]

Comando. Establece la posición actual del cursor gráfico. Es posible, además, establecer las dimensiones de la ventana para gráficos, si se proporcionan las coordenadas opcionales izq, der, arr y abajo. Una llamada a MODE restablece las dimensiones.

```
CLS:BORDER 13
LABEL BUCLE
    ORIGIN 0,0,50,590,350,50
    DRAW 540,350
GOTO BUCLE
```

**OUT puerto,n**

Comando. Envía el valor n al puerto hardware indicado por puerto.

**PAPER [#canal,]tinta**

Comando. Fija la tinta a utilizar como color de fondo. Si no se indica un canal se aplica sobre el canal #0. **Ver nota en PEN.**

```
MODE 1
INK 1,3 ' color rojo
PAPER 1
CLS
```

**PEEK(direccion)**

Función. Devuelve el contenido del byte de memoria en dirección.

```
' Imprime los 5 bytes de un numero real
N! = PI
FOR I=0 TO 4
    PRINT HEX$(PEEK(@N!+I),2);" ";
NEXT
```

**PEN [#canal,]tinta**

Comando. Fija tinta como el color de dibujo para el canal indicado (#0 por defecto).

```
MODE 1
INK 2,3 ' color rojo
PEN 2
PRINT "HOLA MUNDO"
```

**NOTA:** Los valores de PAPER y PEN no se aplican de inmediato, se almacenan en las variables del Firmware y se mandan desde la rutina llamada por las interrupciones al hardware una vez por “frame”. Si se cambian estos valores desde una rutina llamada por EVERY o AFTER muy probablemente no surta efecto.

**PI**

Función. Devuelve el valor real 3.14159265

**PLOT x,y[,tinta[,modo]]**

Comando. Desplaza el cursor gráfico a la posición x e y y dibuja un punto. Si se indica una tinta queda establecida como el color activo. Moves graphics cursor and plots colour from current position. El cuarto parámetro indica el modo o máscara a aplicar entre cada punto de la línea y el fondo, con los valores que se indican a continuación:

Valor	Modo
0	Fill (normal)
1	XOR (OR eXclusivo)
2	AND
3	OR

## PL0TR x,y[,tinta[,modo]]

Comando. Su funcionamiento es igual a PLOT salvo porque x e y son posiciones relativas a la posición actual del cursor gráfico y no posiciones absolutas.

## POKE dirección,n

Comando. Escribe en la posición de memoria dirección el valor (byte) n. Si n es mayor que 255 el valor se trunca.

CLS

```
SUB MEMCOPY(org, dest, n)
  FOR I=0 TO n
    byte = PEEK(org+I)
    POKE dest+I,byte
  NEXT
END SUB
```

A\$ = "HOLA MUNDO"

B\$ = ""

```
CALL MEMCOPY(@A$,@B$,11) ' 10 caracteres mas byte de longitud
PRINT B$
```

## POS(#canal)

Función. Devuelve la posición actual en X del cursor de texto para el canal indicado (#0 por defecto).

MODE 1

```
PRINT POS(#0), VPOS(#0)
```

## PRINT [#canal,][lista de elementos]

Comando. PRINT es un comando muy versatil y con múltiples opciones. Por ello, queda fuera del alcance de este documento y se invita al lector a consultar las obras listadas en el capítulo sobre Referencias. En cualquier caso, ABASC **no soporta el uso de patrones de formato mediante USING**.

## RAD

Comando. Establece que las funciones que devuelven grados den los resultados en radianes. Es el comando contrapuesto a DEG.

```
DEG
PRINT SIN(90.0)
RAD
PRINT SIN(90.0)
```

## **RANDOMIZE [n]**

Comando. La implementación soportada por ABASC difiere un poco del comportamiento habitual de este comando en Locomotive BASIC. Si se usa RANDOMIZE sin parámetros, ABASC lo interpreta como si se hubiera usado RANDOMIZE TIME. El uso de RANDOMIZE y RND implica el uso de números reales.

```
RANDOMIZE
FOR I=1 TO 20
    PRINT RND
NEXT
```

## **READ lista-de-variables**

Comando. Lee el siguiente dato de los declarados con DATA y lo asigna a la variable correspondiente de su lista. El programador es el responsable de que el tipo de dato actual y el tipo de la variable coincidan.

```
CLS
FOR I=0 TO 5
    READ nom$
    PRINT "Nombre:", nom$
NEXT
END
```

```
DATA "Xavier","Ross","Gada",
DATA "Anabel","Rachel","Elvira"
```

## **READIN lista-de-variables**

Comando. Es equivalente a INPUT #9, es decir, lee datos del fichero de entrada abierto y los asigna a la lista de variables. Actualmente, ABASC no soporta variables reales en este comando.

## **RECORD nombre;lista-de-variables**

Comando. Permite declarar un registro que puede aplicarse a variables de tipo cadena (\$) para crear estructuras de datos. Se invita al lector a consultar el apartado sobre Estructuras con RECORD en la sección Tipos y variables del capítulo Peculiaridades del compilador.

```
DECLARE A$ FIXED 13 ' No es obligatorio, pero reduce el consumo de memoria
RECORD persona; nom$ FIXED 10, edad ' Requiere 13 bytes de memoria
```

```
A$.persona.nom$ = "Juan"
A$.persona.edad = 20
```

## RELEASE canal

Comando. Los sonidos encolados en un determinado canal pueden contener un estado de espera. Este comando libera dichos sonidos. canal es un número entero que indica los canales afectados:

- 1 = canal A
- 2 = canal B
- 4 = canal C

RELEASE 7 'libera los sonidos en los tres canales

## REM texto

Comando. Permite añadir comentarios al texto. Un alias es el simbolo '.

## REMAIN(temporizador)

Función. Desactiva el evento asignado a temporizador(en el rango 0..3) y devuelve cuantos “ticks” quedaban para su activación. Dichos eventos se registran con AFTER o EVERY.

## RENUM nueva-línea, línea-origen, incremento

Comando. En Locomotive BASIC permite reenumerar las líneas de código de un programa en BASIC. En un programa compilado no tiene sentido. ABASC ignora este comando y emite una advertencia si lo encuentra en el código a compilar.

## RESTORE [etiqueta]

Comando. Establece que el siguiente dato a leer con READ sea el primer valor declarado con DATA encontrado tras la etiqueta indicada, sea esta un número de línea o un identificador declarado con LABEL. Si no se especifica ninguna etiqueta el comando establece el primer dato declarado con DATA encontrado en el programa.

```

LABEL BUCLE
FOR N=1 TO 5
    READ A$
    PRINT A$;" ";
    DATA datos,"a leer",una,"y otra",vez
NEXT
PRINT
RESTORE
GOTO BUCLE

```

## RESUME

Comando. Restaura la ejecución de un programa detenido tras un evento de error manejado por ON ERROR GOTO. Como ABASC implementa ON ERROR GOTO de forma algo diferente, ignorará este comando y emitirá una alerta si aparece como parte del código a compilar.

**RETURN**

Comando. Continúa la ejecución del programa en la siguiente instrucción al último GOSUB ejecutado.

**RIGHT\$(cadena,n)**

Función. Devuelve los primeros n caracteres comenzando la cuenta por la derecha de cadena.

```
PRINT RIGHT$("AMSTRAD", 3)
```

**RND[(0)]**

Función. Devuelve un número pseudoaleatorio en el rango [0.0 - 1.0]. Si se llama con el parámetro 0 (RND(0)) devuelve, de nuevo, el último número que se generó. El uso de RANDOMIZE y RND implica el uso de números reales.

```
RANDOMIZE
FOR I=1 TO 20
    PRINT RND
NEXT
```

**ROUND(x[,n])**

Función. Redondea el número real x a la posición decimal indicada por n (0 por defecto).

```
FOR I=0 TO 4
    PRINT ROUND(PI, I)
NEXT
PRINT ROUND(PI,-3)
```

**RUN [etiqueta | fichero]**

Comando. En BASIC este comando permite ejecutar desde el principio el programa actualmente en memoria (sin argumentos), ejecutar un programa en memoria desde la etiqueta indicada o carga un programa desde fichero y lo ejecuta desde el principio. ABASC solo soporta las dos primeras formas. En ambas, ejecuta un CLEAR antes de saltar al principio del programa o la etiqueta indicada para asegurar cierta consistencia entre ejecuciones.

**SAVE fichero[,tipo][,dirección,tamaño[,entrada]]**

Comando. En BASIC permite grabar un programa a disco o cassette. ABASC solo permite grabar una región de memoria como fichero binario. Por tanto, el tipo del fichero siempre se considera (y debe indicarse así si se van a utilizar el resto de parámetros) B. Como referencia, los tipos permitidos por la instrucción en BASIC son:

- A - Texto (ASCII)
- P - Fichero protegido
- B - Binario

El resto de parámetros opcionales son:

Parámetro	Función
dirección	Dirección de memoria desde donde comenzar el volcado.
tamaño	Total de bytes que se deben volcar al fichero.
entrada	Dirección donde empezar la ejecución del binario si se carga con RUN.

```
MODE 1
PAPER 3
CLS
SAVE "pantalla.bin",B,&C000,&4000
PAPER 0
CLS
LOAD "pantalla.bin"
```

### SGN(x)

Función. Devuelve -1 si x es menor que 0, devuelve 0 si x es igual a 0 o devuelve 1 si x es mayor que cero.

```
PRINT SGN(PI)
```

### SHARED variable | array [,variable | array]

Este comando proviene del Locomotive BASIC 2 Plus. Permite que desde una rutina (SUB o FUNCTION) se referencie y utilice una variable global del programa. Si el nombre de la variable termina en corchetes, se interpreta que la variable es un array declarado con DIM.

```
DIM vec(3)

SUB setvec()
    SHARED vec[]
    vec(0) = 1
    vec(1) = 2
    vec(2) = 3
END SUB
```

```
call setvec()
```

### SIN(x)

Función. Devuelve el seno de x. Implica el uso de números reales.

## **SOUND canal,perido-ono,duracion,volumen,env,ent,ruido**

Comando. SOUND es uno de los principales puntos fuertes de Locomotive BASIC comparado con el resto de las versiones BASIC de la época. Es un comando muy versátil que proporciona un acceso muy amplio al chip de audio de los Amstrad CPC. Por tanto, el lector hará bien en recurrir a los libros de la sección Referencias para aprender todos los entresijos de este comando.

```
ENV 2,127,0,0,127,0,0,127,0,0,127,0,0,127,0,0
SOUND 1,1000,0,12,2
SOUND 2,900,0,12,2
```

## **SPACE\$(n)**

Función. Devuelve una cadena de texto con tantos espacios en blanco como los indicados por n.

## **SPEED INK t1,t2**

Comando. INK y BORDER permiten especificar dos colores entre los que se alternará. SPEED INK permite especificar cuanto tiempo estará visible cada uno de los dos colores. Los tiempos t1 y t2 se indican en “frames” (50 por segundo).

```
SPEED INK 150,50 ' 3 segundos y 1 segundo
BORDER 0,1
```

## **SPEED KEY espera,repetición**

Comando. Si se mantiene pulsada una tecla, esta comenzará a repetirse cuando se supera su tiempo de espera, cada vez que venza el tiempo de repetición. Los tiempos deben darse en “frames” (50 por segundo) en un rango de 1 a 255.

## **SPEED WRITE n**

Comando. Cambia la velocidad (en baudios) a la que se escribe en cassette. n puede ser 1 (2000 baudios) o 0 (1000 baudios).

## **SQ canal**

Función. Permite comprobar el número de entradas libres en la cola para el canal indicado (1,2 o 4). Determina si dicho canal está activo y, en caso contrario, por qué la entrada activa de la cola (si la hay) está en espera. El resultado es un entero que codifica la información como sigue:

- Los bits 0, 1 y 2 indican el número de huecos libres en la cola.
- Los bits 3, 4 y 5 indican el estado de sincronización de la primera nota en la cola.
- El bit 6 se activa si la primera nota está en espera.
- El bit 7 se activa si el canal está activo ahora mismo.

```
SOUND 65,100,100
PRINT BIN$(SQ(1),8) ' debe imprimir 01000011
```

**SQR(x)**

Función. Devuelve la raíz cuadrada de x. Implica el uso de números reales.

**STOP**

Comando. En Locomotive BASIC detiene la ejecución del programa y devuelve el control al interprete. El usuario puede retomar la ejecución con CONT. Puesto que no tiene mucho uso en un programa compilado, ABASC reutiliza esta instrucción para formar un reinicio de la máquina (CALL 0).

**STR\$(x)**

Función. Devuelve una cadena con el número x convertido en texto.

```
PRINT "PI = " + STR$(PI)
```

**STRING\$(n,carácter)**

Función. Devuelve una cadena de texto con el carácter indicado repetido n veces.

```
MODE 1
LOCATE 1,10
PRINT STRING$(40,250)
```

**SUB [(parámetros)] [ASM]**

Comando. Proveniente de Locomotive BASIC 2 Plus, SUB permite declarar procedimientos con parámetros. Debe utilizarse CALL para llamar a un procedimiento declarado con SUB. El procedimiento debe declararse antes de que aparezca en el código una llamada al mismo. Si se utiliza la cláusula ASM, ABASC entiende que el cuerpo del procedimiento va a ser mayoritariamente código en ensamblador que no va a usar el mecanismo de memoria temporal, por lo que no se encarga de apilarlo y restaruralo después de cada llamada.

Se recomienda al programador leer las secciones Procedimientos y Funciones y Uso de código ensamblador del capítulo Peculiaridades del compilador para obtener más información sobre el tratamiento de los parámetros o el soporte a la recursividad.

```
SUB miUSING(n,long)
  ' Imprime el número N con una LONG fija, rellenando
  ' con 0 los espacios sobrantes a la izquierda.
  n$ = STR$(n)
  text$ = STRING$(long,48) ' rellena con 0
  digitos = LEN(n$)
  inicio = long - LEN(n$) + 1
  MID$(text$,inicio,digitos)=n$
  PRINT text$
END SUB
```

```
num=1234
CALL miUSING(num,8)
```

## SYMBOL carácter,valor1,valor2,...,valor8

Comando. Redefine el símbolo indicado por el número carácter. Dicho número debe estar disponible para redefinirse (ver SYMBOL AFTER). Cada carácter viene representado por una matriz de 8x8 píxeles. Los siguientes 8 valores definen cada fila del carácter. El valor es la suma de píxeles de esa línea que deben pintarse con el color de la tinta actual. Cada pixel de la línea tiene un valor numérico tal y como sigue:

pixel 1	pixel 2	pixel 3	pixel 4	pixel 5	pixel 6	pixel 7	pixel 8
128	64	32	16	8	4	2	1

SYMBOL AFTER 240

SYMBOL 240,&00,&00,&74,&7E,&6C,&70,&7C,&30

SYMBOL 241,&7E,&FD,&80,&80,&80,&80,&40,&00

SYMBOL 242,&00,&00,&08,&00,&00,&00,&00,&00

SYMBOL 243,&00,&00,&00,&00,&10,&0C,&00,&00

SYMBOL 244,&60,&F8,&FC,&FC,&FC,&FC,&FC,&FC

SYMBOL 245,&00,&00,&60,&60,&30,&30,&00,&00

SYMBOL 246,&00,&00,&00,&00,&0C,&0C,&00,&00

SYMBOL 247,&FC,&FC,&EC,&CC,&CC,&CC,&00,&00

SYMBOL 248,&00,&00,&00,&00,&00,&00,&EE,&EE

MODE 0

PRINT CHR\$(22)+CHR\$(1) ' Modo transparente de escritura ON

LOCATE 5,2:PEN 11:PRINT CHR\$(240);

LOCATE 5,2:PEN 1:PRINT CHR\$(241);

LOCATE 5,2:PEN 8:PRINT CHR\$(242);

LOCATE 5,2:PEN 3:PRINT CHR\$(243);

LOCATE 5,3:PEN 10:PRINT CHR\$(244);

LOCATE 5,3:PEN 6:PRINT CHR\$(245);

LOCATE 5,3:PEN 11:PRINT CHR\$(246);

LOCATE 5,4:PEN 9:PRINT CHR\$(247);

LOCATE 5,4:PEN 3:PRINT CHR\$(248);

PRINT CHR\$(22)+CHR\$(0) ' Modo transparente de escritura OFF

## SYMBOL AFTER n

Comando. Fija el número del carácter a partir del cuál se pueden redefinir. n debe ser un valor entre 1 y 256. Por defecto, los programas tienen disponibles los caracteres desde el valor 240 al 255. El valor disponible en un programa compilado será el valor más bajo de los utilizados en SYMBOL AFTER si existen múltiples llamadas.

ABASC reserva 8 bytes por cada carácter que puede redefinirse. Si no se va a utilizar esta capacidad es recomendable comenzar el programa con SYMBOL AFTER 256, lo que evitará que se reserve memoria. Para un ejemplo, ver SYMBOL.

**TAG [#canal]**

Comando. Redirige la salida de texto asociada a canal (#0 por defecto) para utilizar como el cursor gráfico en vez del cursor de texto. Esto permite mezclar texto con gráficos o desplazar el texto por píxeles en vez de por bloques de 8x8.

```
MODE 2
BORDER 9
INK 0,12: INK 1,0
LABEL BUCLE
TAG
FOR n=1 TO 100
    MOVE 200+n,320+n
    IF n<70 THEN
        PRINT"Hola";
    ELSE
        PRINT"Adios";
    END IF
NEXT
GOTO BUCLE
```

**TAGOFF [#canal]**

Comando.Desactiva el uso del cursor gráfico para el canal de texto indicado (#0 por defecto). Ver TAG.

**TAN(x)**

Función. Devuelve la tangente del ángulo x. Implica el uso de números reales.

```
PRINT TAN(45)
```

**TEST(x,y)**

Función. Devuelve el valor de la tinta en la posición de pantalla x e y.

```
MODE 1
PRINT TEST(320,200)
PLOT 320,200,1
PRINT TEST(320,200)
```

**TESTR(x,y)**

Función. Igual que TEST pero siendo x e y posiciones relativas y no absolutas.

## TIME[ (n) ]

Función. Devuelve el tiempo transcurrido desde el encendido de la máquina. Mide el tiempo en pasos de 1/300 segundos. Requiere que las interrupciones estén activas, por lo que DI y ciertas operaciones de disco/cinta harán que deje de contarse el tiempo. El valor devuelto es un número real.

ABASC permite una segunda forma de uso en la que TIME se comporta como un **comando**. Es este caso, es posible especificar un valor entero entre paréntesis y ese valor se fijará como el nuevo valor de TIME.

CLS

T! = TIME        ' Podria usarse TIME(0)

FOR i=1 to 10

    FOR j=1 to 1000

        s = 1000 + j

    NEXT j

    PRINT ".";

NEXT i

PRINT " FIN!"

PRINT "Tiempo="; (TIME-T!)/300.0; "s" ' Si se uso TIME(0) no hace falta restar

Por último, si ABASC detecta que se está convirtiendo el valor devuelto por TIME a un entero, ejecuta una optimización en la llamada para evitar el uso de números reales. Sin embargo, el programador debería tener cuidado al usar TIME de este modo, ya que el valor da una vuelta entera cada 3 segundos debido a la menor precisión de los números enteros.

TIME(0)

FOR I=0 TO 20

    FRAME

    PRINT CINT(TIME)

NEXT

## TROFF

Comando. Desactiva la impresión de trazas. ABASC ignora este comando si lo encuentra como parte del código y emite una advertencia. Ver TRON.

## TRON

Comando. En Locomotive BASIC permite emitir trazas según se interpreta un programa. ABASC ignora este comando si lo encuentra como parte del código y emite una advertencia.

## UNT(n)

Comando. Convierte un valor sin signo (como una dirección de memoria) en el rango 0..65535 en un entero con signo en el rango -32768..+32767.

PRINT UNT(&FF66) ' debe imprimir el valor -154

**UPPER\$(cadena)**

Función. Devuelve cadena con todos sus caracteres pasados a mayúsculas.

```
C$="AmSTRaD"
PRINT LOWER$(C$)
PRINT UPPER$(C$)
```

**VAL(cadena)**

Función. Devuelve el primer **número entero** encontrado en cadena. Por tanto, y a diferencia del intérprete de BASIC en las máquinas Amstrad CPC, VAL no se puede usar para extraer un número real de una cadena de texto.

```
PRINT VAL("15") + 15
```

**VPOS(#canal)**

Función. Devuelve la posición actual en Y del cursor de texto para el canal indicado (#0 por defecto).

```
MODE 1
PRINT POS(#0), VPOS(#0)
```

**WAIT puerto,mascara[,inversion]**

Comando. Detiene la ejecución hasta que se lee un valor esperado desde el puerto de entrada/salida especificado. El comando realiza una operación de **AND** con la mascara indicada y una operación de **XOR** con el valor de inversion (si se suministra). La ejecución solo continúa si el resultado obtenido es distinto de 0.

```
WAIT &FF34,20,25
```

**WEND**

Comando. Marca el final de un bucle WHILE.

**WHILE condición**

Comando. Marca el inicio de un bucle del que solo se sale cuando condición es cierta.

```
CLS
PRINT "Espera de 10 segundos": T! = TIME + 3000
WHILE TIME<T!
    SOUND 1,0,100,15
WEND
SOUND 129,40,30,15
```

**WIDTH n**

Comando. Especifica el ancho en caracteres máximo a soportar por la impresora. ABASC no soporta este comando y emitirá una advertencia si lo encuentra en el código a compilar.

**WINDOW [#canal,]izq,derecha,arriba,abajo**

Comando. Define una nueva ventana de texto asociada a canal que debe estar en el rango #0..#7 (#0 por defecto).

```
MODE 1
WINDOW #1,1,40,20,25
WINDOW #2,2,39,21,24
PAPER 0
PAPER #1,1
PAPER #2,2
CLS#0
CLS#1
CLS#2
```

**WINDOW SWAP canal1,canal2**

Comando. Intercambia las características de las ventanas de texto indicadas por canal1 y canal2.

```
MODE 1
WINDOW #1,1,40,20,25
PAPER 0
PAPER #1,2
CLS#0
CLS#1
WINDOW SWAP 0,1
PRINT "VENTANA 0"
```

**WRITE [#canal],dato1,dato2,...**

Comando. En Locomotive BASIC escribe los valores proporcionados en el canal indicado (#0 por defecto). ABASC ignora el valor del canal y siempre lo considera #9, el canal para operaciones con ficheros. Por tanto, WRITE se puede utilizar para escribir en un fichero mientras que READIN serviría para leer los datos de vuelta. Los números reales no están soportados, solo se pueden escribir datos enteros o cadenas de texto.

```
A=15
NOM$="Juan"
OPENOUT "DATOS.TXT"
WRITE #9,NOM$,A
CLOSEOUT
```

**XPOS**

Función. Devuelve la posición en X del cursor gráfico.

```
MODE 1
PRINT XPOS;YPOS
MOVE 320,200
PRINT XPOS;YPOS
```

## YPOS

Función. Devuelve la posición en Y del cursor gráfico. Ver XPOS.

## ZONE n

Comando. Cambia la anchura (13 por defecto) de la zona de escritura utilizada por PRINT cuando se separan elementos con comas.

```
CLS
PRINT "A","B"
ZONE 4
PRINT "A","B"
```

---

# Apéndice I: Depurar programas compilados

Depurar programas generados por un compilador cruzado puede ser una tarea compleja, ya que la máquina que ejecuta el código es diferente de la máquina donde se desarrolló. Afortunadamente, los emuladores pueden simplificar significativamente este proceso. Por ejemplo, **WinApe** y **Retro Virtual Machine** permiten configurar un entorno de depuración eficaz.

## Comprobación del código BASIC

**WinApe** ofrece una forma conveniente de “pegar” código BASIC y ejecutarlo. Esto nos permite comparar los resultados entre el intérprete de BASIC y nuestro código compilado. Naturalmente, para que nuestro código funcione, no podremos utilizar las opciones provenientes del Locomotive BASIC 2 (como FUNCTION, SUB, IF multilinea, etc.). Sin embargo, sí podremos usar las siguientes características:

- Código sin números de línea
- Código dividido en varios archivos

Al compilar con ABASC, el primer paso lo realiza el preprocesador. Con la opción `--verbose` activada, genera un archivo intermedio con la extensión `.BPP`, en el que se añaden números de línea y se añade el código proveniente de los archivos adicionales referenciados mediante `CHAIN MERGE`.

Para pegar código en **WinApe**, sigue estos pasos:

1. Selecciona el código deseado en tu editor favorito y elige la opción Copiar.
2. En **WinApe**, ve al menú File y selecciona Paste.
3. Si estás pegando una gran cantidad de código, activa Settings > High Speed para acelerar el proceso. Recuerda volver a Normal Speed una vez completado el pegado.

## Depuración paso a paso de nuestro código

No es posible depurar código BASIC paso a paso, pero sí podemos depurar el código ensamblador generado por el compilador. Como parte del proceso de compilación, ABASC genera un archivo intermedio con extensión `.ASM`. Este archivo utiliza una sintaxis compatible con **WinApe** y **Retro Virtual Machine 2.0**.

En **Retro Virtual Machine**, podemos activar las herramientas de depuración siguiendo estos pasos:

1. Abrir nuestra máquina Amstrad CPC (464 o 6128).
2. Presionar en el menú de hamburguesa en la esquina superior izquierda.
3. Activar la opción `Developer Mode`.

Aparecerá en la barra superior de iconos un botón con el símbolo de un martillo. Al hacer clic sobre él, se desplegará un submenú con varias herramientas; seleccionaremos la última, la consola de **Retro Virtual Machine**. Desde esta consola, podremos navegar por los directorios de nuestra máquina y cargar nuestro código de la siguiente manera:

- `ls` — Lista el contenido del directorio actual.
- `cd` — Cambia de directorio.
- `asm` — Ensambla el archivo `.ASM` especificado.

Este método permite cargar nuestro programa en un entorno de prueba mucho más rápido que usando archivos `.DSK` y el soporte de disco. Una vez que el programa está en memoria, se puede ejecutar mediante el comando:

```
CALL &170
```

Además, después de ensamblar el código con **Retro Virtual Machine**, es posible listar todos los símbolos (etiquetas de línea, nombres de variables, etc.) en la consola con el comando `symbols`. Esto nos permite establecer puntos de parada (breakpoints) en cualquier posición de memoria usando:

```
break dirección-de-memoria
```

Para borrar todos los puntos de parada, basta con ejecutar:

```
break -x
```

Este proceso de depuración requiere cierta familiaridad con el código ensamblador. En la sección de Referencias se incluyen libros y materiales que pueden servir como guía de aprendizaje.

Finalmente, se recomienda al lector consultar la documentación oficial de **WinApe** y **Retro Virtual Machine** para explorar más opciones de depuración y aprovechar al máximo las herramientas que ofrecen estos emuladores.

## Apéndice II: Ampliando el compilador

Una de las grandes ventajas de ABASC es que, al estar escrito en Python, resulta sencillo **ampliar y modificar sus funciones**. El código fuente se organiza en los siguientes archivos principales:

- **abasc.py – Fichero principal:** Gestiona las distintas opciones del compilador y ejecuta la compilación paso a paso.
- **baspp.py – Preprocesador:** Añade números de línea e inserta cualquier fichero de código adicional referenciado mediante CHAIN MERGE. Si se activa la opción `--verbose`, genera un fichero intermedio con extensión `.BPP`.
- **baslex.py – Analizador léxico:** Recorre el código fuente y genera la lista de tokens correspondiente. Con `--verbose`, produce un fichero intermedio con extensión `.LEX`.
- **basparse.py – Analizador sintáctico:** Procesa la lista de tokens, verifica la sintaxis del programa y genera una representación intermedia del código en forma de Árbol de Sintaxis Abstracta (AST). Con `--verbose`, se genera un fichero intermedio `.AST`.
- **emitters/cpcemitter.py – Generador de código ensamblador:** Toma el AST generado por el analizador sintáctico y produce el código ensamblador equivalente. El resultado se guarda en un fichero `.ASM`, que luego será ensamblado por **ABASM** para producir el binario final.
- **emitters/cpcrt.py – Runtime del compilador:** Contiene rutinas en ensamblador llamadas por el código generado por `cpcemitter.py`.

Siempre que se realicen cambios en cualquiera de estos archivos, es recomendable comprobar que no se han introducido errores evidentes. Esto se puede hacer ejecutando los siguientes comandos desde el directorio donde se encuentra `abasc.py`:

- **Comprobación de tipos:**

```
mypy . --explicit-package-bases
```

- **Pruebas unitarias:**

```
python3 -m unittest -b
```

Finalmente, el directorio `examples` incluye varios programas de ejemplo que pueden compilarse y servir también para realizar pruebas y experimentos con el compilador.

---