Vircon32: Aprender el lenguaje C

Documento con fecha 2023.01.23 Escrito por Carra

¿Qué es esto?

Este documento es una guía rápida para empezar a programar en lenguaje C si no lo conocías antes. Esta guía pretende dar una comprensión básica de la estructura de un programa, características básicas del lenguaje C y mostrar lo que se necesita para quien quiera crear programas en C para la consola Vircon32. Aún así, se espera que conozcas conceptos básicos de programación, como funciones, variables, arrays, etc.

¡Algo que deberías saber!

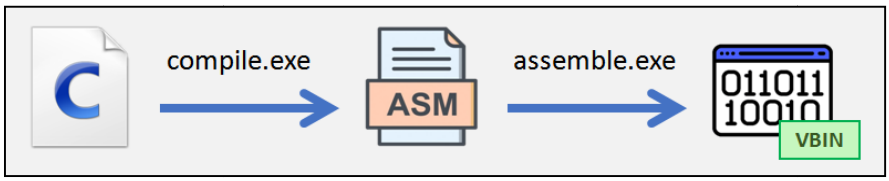
Esta guía sólo enseña las características básicas de C: las mínimas necesarias para crear programas en C. Por claridad, esta guía omite deliberadamente varios de los aspectos más avanzados del lenguaje. Además, debe tenerse en cuenta que existen diferencias menores entre el lenguaje C estándar y el usado en Vircon32. Esta guía se centra en la versión Vircon32. Para más detalles, lee la guía del compilador de C de Vircon32.

Índice

Este documento está organizado en secciones. Cada una mostrará progresivamente más detalles sobre el lenguaje C utilizado en Vircon32 y su compilador.

Introducción

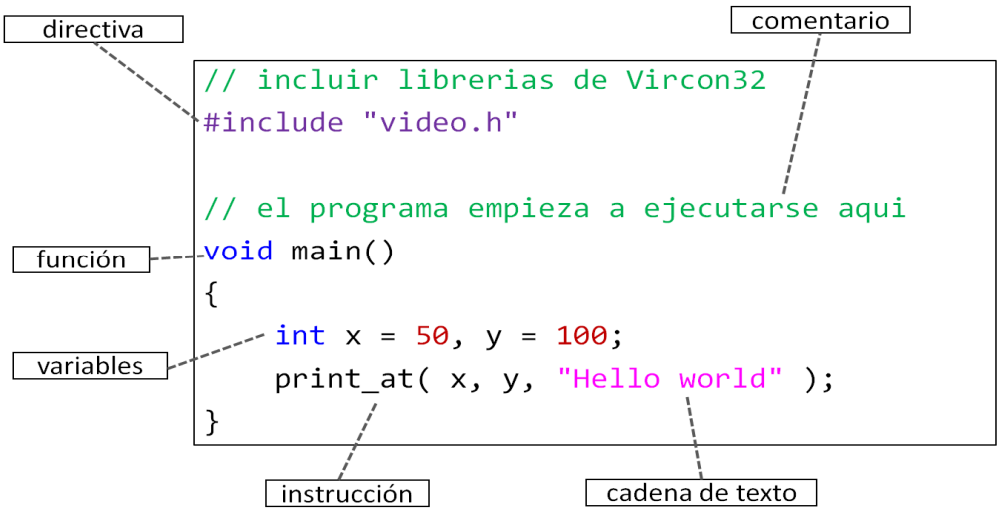
C es un lenguaje compilado. Esto significa que, antes de poder ejecutarse, el código que escribas debe transformarse en instrucciones máquina que la CPU de Vircon32 sea capaz de ejecutar. En esta consola el proceso sería así:



Una vez creado ese archivo binario con las instrucciones de la CPU, tendrás que empaquetar ese programa en un archivo .v32 junto con las imágenes y sonidos que necesite. Pero esta guía se centra sólo en escribir los programas. Puedes leer el proceso completo para crear juegos en las otras guías de Vircon32.

Estructura de un programa en C

Un tutorial de un lenguaje siempre debe empezar con un simple programa "hola mundo", así que aquí lo tienes. Aquí puedes ver la estructura más básica de un programa en C.



La función main

La función main es una función especial que debe existir siempre, ya que la ejecución de nuestro código comienza siempre al principio de esta función. En este compilador main no toma argumentos ni devuelve ningún valor, por lo que debe declararse con este prototipo:

```
void main()   // o bien esto o bien: void main( void )
{
    // ...
}
```

Comentarios

Los comentarios son una forma de incluir en el programa nuestras explicaciones sin que el compilador trate de interpretarlas (lo que provocaría errores). Es decir, los comentarios no forman parte del programa en sí. Existen 2 tipos de comentarios, y funcionan de la misma forma que en el lenguaje C estándar:

Comentario de línea

Estos comentarios empiezan con `//`, y terminan al final de la línea.

```
int x = 7 + 5;    // comentario que afecta solo a esta linea
int y = x;
```

Comentario de bloque

Un comentario de bloque comienza con `/*`, y no terminará hasta que lo cerremos con `*/`, incluso si se hace en una línea diferente.

```
/* este comentario invalida la linea de debajo
int x = 7 + 5;
*/

int y;
```

Tipos de datos

Todos los valores y variables que maneja el lenguaje pertenecen a algún tipo de dato que indica cómo interpretar ese valor. Aquí veremos qué tipos de datos podemos usar.

Tipos básicos

Vircon32 organiza la memoria en palabras de 32 bits porque en esta consola la unidad mínima de memoria es la palabra (y no el byte), por lo que debemos tener en cuenta que:

- Todos los tamaños de tipos de datos se miden en palabras y no en bytes
- Todas las direcciones de memoria y offsets se dan en palabras y no en bytes

Los 4 tipos de datos básicos en este compilador son:

```
int     float     bool     void
```

Todos estos tipos tienen tamaño 1 (es decir, 32 bits). El tipo void es especial, y sólo puede usarse en su forma básica para indicar que una función no devuelve ningún valor.

Tipos derivados

Podemos crear nuevos tipos utilizando los tipos básicos en arrays y punteros. Esto puede hacerse varias veces, por ejemplo:

```
int*   void**   float[3][5]   void*[4]
```

Podemos usar void para crear punteros, pero no arrays (no puede haber valores de tipo void). C permite punteros a void: indican una dirección de memoria donde la información se lee o almacena como simples bits, sin interpretarla (void indica la ausencia de tipo).

Tipos compuestos

Podemos agrupar varios tipos de datos en un tipo mayor, creando estructuras y uniones. Cada miembro de estos grupos es un campo al que se accede por nombre.

```
// declarar tipo Point
struct Point
{
    int x, y;
};

// declarar tipo Word
union Word
{
    int AsInteger;
    void* AsPointer;
};

// usar estructuras
Point P1,P2;
P1.x = 10;
P1.y = -7;
P2 = P1;

// usar uniones
Word W;
W.AsInteger = 0xFF110AF;
int* Pointer = W.AsPointer;
```

Las estructuras y uniones, a su vez, pueden contener otras estructuras y uniones. También pueden incluirse a sí mismas, pero sólo con punteros por razones obvias.

Enumerados

Podemos definir una serie de constantes enteras y agruparlas en su propio tipo. El lenguaje C permite esto mediante enumerados. En general estos valores se tratan como enteros y se pueden utilizar de la misma forma, pero su tipo es más restrictivo.

Si no se especifican valores, la primera constante tendrá un valor numérico de 0 y cada constante siguiente tendrá un valor igual a la constante anterior más 1.

```
// declarar tipo Semaphore
enum Semaphore
{
    Red = 1,
    Yellow,   // tiene valor 2
    Green,    // tiene valor 3
};

// estas operaciones son correctas
Semaphore S1,S2;
S1 = Red;
S2 = S1;
int Value = Yellow + Green;  // se puede convertir enum a int

// estas asignaciones producirian errores!
S1 = Green - Red;
S2 = 1;    // mismo valor que Red, pero no puede convertirse int a enum
```

Valores literales

En nuestros programas podemos usar valores constantes que escribimos literalmente. Según el tipo de dato que represente cada uno de estos valores, tenemos diferentes notaciones y representaciones numéricas. En este compilador existen las siguientes:

```
-15;     // int en decimal
0xFF1A;  // int en hexadecimal
0.514;   // float
true;    // bool
'a';     // int como caracter (no existe char)
"hi!";   // int[4] (cadena de 3 caracteres + terminacion 0)
NULL;    // puntero nulo
```

Los valores de los literales booleanos son: true = 1, false = 0.

Variables

Además de datos constantes también podemos manejar variables: son direcciones de memoria donde se almacena el valor que varía. A este valor se accede usando un nombre que identifica la variable, y un tipo de dato que interpreta el valor almacenado.

En C, tus variables sólo pueden almacenar el tipo de valores con el que se declaran (por ejemplo, enteros). En otros lenguajes puedes declarar una variable sin tipo e ir almacenando diferentes valores en ella (números, cadenas, etc). Esto no se permite en C.

Declarar variables

Una variable se declara con un tipo y un nombre, y opcionalmente puede inicializarse con un valor:

```
float Speed = 2.5;
bool Enabled;
```

En este compilador los tipos siempre se mantienen separados del nombre, en este caso: <tipo> <nombre>. Por ejemplo, en este compilador un array se debe declarar así:

```
// este es un array de 5 floats
float[5] BallSpeeds;

// este es un array con 20 posiciones, y cada una es un array de 10 ints
int[20][10] LevelBricks;
```

Declaraciones múltiples

También es posible declarar varias variables del mismo tipo en una sola declaración, separándolas con comas. Las variables declaradas así siempre son todas del mismo tipo.

```
// declarar 3 punteros a int
int* ptr1 = &number, ptr2 = ptr1, ptr3;
```

Listas de initialización

Las matrices y estructuras pueden inicializarse con una lista de múltiples valores. Estas listas también pueden anidarse para tipos más complejos, como se ve en estos ejemplos:

```
// inicializar una estructura
struct Point
{
    int x, y;
};

Point P = { 3, -7 };

// aqui usamos listas anidadas para un array de estructuras
Point[3] TrianglePoints = { {0,0}, {1,0}, {1,1} };

// para arrays de int tambien podemos usar una cadena en vez de una lista
int[10] Text = "Hello";  // cuidado, C añade un caracter extra (0) de terminacion
```

Ámbito de una variable

En lenguaje C existen 2 clases básicas de variables: locales y globales.

- Una variable local se declara en el cuerpo de una función y sólo es accesible dentro de él (porque se almacena en la pila, que cambia durante la ejecución).

- Una variable global se declara fuera de las funciones y es accesible a todo el programa después de su declaración porque su dirección es fija.

Punteros

No hacen falta punteros para crear juegos básicos, pero quizá quieras tener una idea de lo que son. Un puntero es sólo una variable que contiene una dirección de memoria (o sea, un número que designa una posición en la memoria). Por ejemplo podríamos tener:

```
int Variable = 7;  // digamos que Variable se guarda en la posicion de memoria 1250
int* Pointer = &Variable;  // Pointer ahora contiene el numero 1250
```

Este puntero se declara con tipo int*. Esto significa que la posición de memoria a la que apunte, se interpretará como un entero. Así, podemos leer o modificar el valor almacenado en nuestra variable de forma indirecta, usando el puntero como intermediario.

¿Por qué son útiles los punteros?

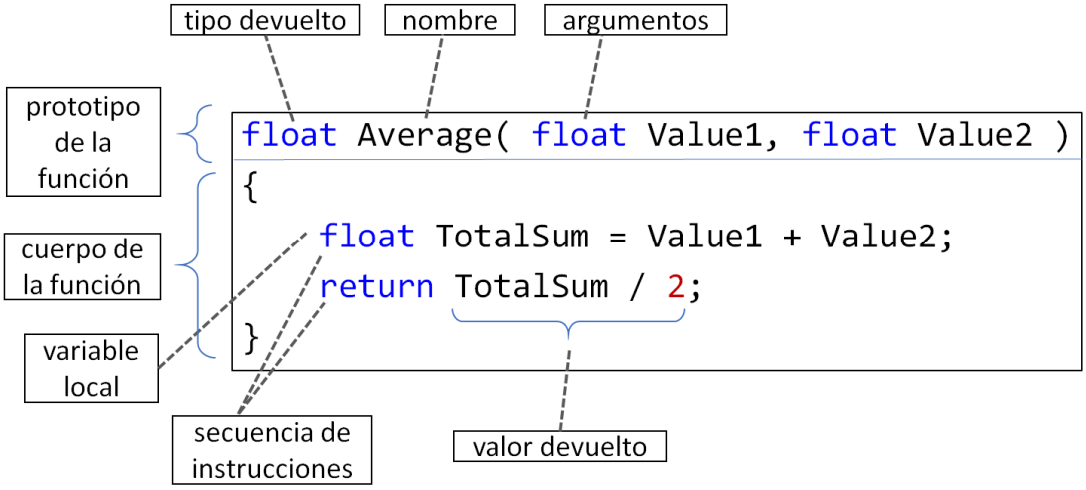
Los punteros se pueden usar para muchas cosas, pero un ejemplo típico es pasar un puntero a una función para permitirle manipular datos externos. Por ejemplo, puedes tener una función que devuelva 2 valores, pasándole 2 punteros a variables externas. O, si tienes una estructura o array demasiado grande para pasarla como parámetro, puedes trabajar con un puntero a ella y leer/modificar la estructura original.

El puntero NULL

C también tiene un valor literal especial, NULL, usado en punteros. NULL existe para poder tener una forma de indicar que un puntero actualmente no está apuntando a nada. El valor de NULL suele designar alguna dirección de memoria no válida de forma que, si intentamos leer/escribir desde ella, se producirá una violación de memoria. Esto ayuda a detectar errores en los programas.

Funciones

En lenguaje C no está permitido ejecutar ninguna instrucción (que no sean meras declaraciones) fuera de una función. Para poder ejecutar código en nuestro programa, necesitaremos declarar funciones que contengan las instrucciones a ejecutar. Las funciones en C se declaran de esta forma:



El cuerpo de una función puede contener múltiples instrucciones que se ejecutarán de manera secuencial.

Retorno de funciones

Dentro del cuerpo de una función, podemos usar `return` para salir de la función en cualquier momento. La ejecución volverá al punto desde el que se llamó a esa función.

Return también se utiliza para devolver un valor (cuando el tipo devuelto por la función no es void). En ese caso, return debe usarse con un valor de tipo compatible.

```
// funcion que no devuelve un valor
void DoNothing()
{
    return;
}

// function que devuelve un puntero
int* FindLetterA( int* Text )
{
    while( Text )
    {
        if( *Text == 'A' )
          return Text;

        Text++;
    }

    return NULL;
}
```

Limitaciones de las funciones

Una diferencia importante con el C estándar es que, por limitaciones de este compilador, las funciones no pueden recibir parámetros ni devolver valores de tamaño distinto de 1.

Es decir: no pueden usar arrays, uniones o estructuras (a menos que su tamaño sea de una sola palabra). En vez de eso, deben operar con punteros a los mismos.

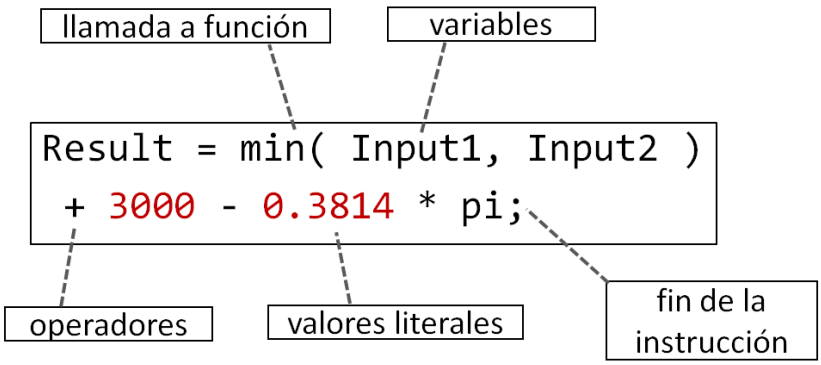
Sin embargo, se puede "pasar un array" como parámetro cuando el array decae a puntero, de la misma forma que en el lenguaje C estándar. Por ejemplo, podemos hacer esto:

```
// funccion que suma N enteros
int SumValues( int* Values, int N )
{  // ...  }


// array de 10 enteros
int[10] Values;

// sumamos los primeros 5 valores del array
int Sum = SumValues( Values, 5 );
```

En este caso, el "decaimiento" hace que el array Values se evalúe como un puntero a su primer elemento. Por eso es aceptado por una función que espera un puntero a entero.

Expresiones

Las expresiones son operaciones que podemos realizar combinando variables y valores literales. Estas combinaciones se pueden hacer aplicándoles funciones u operadores, hasta obtener como resultado un único valor final. En esta expresión de ejemplo tenemos:



Operadores

En su mayor parte los operadores representan operaciones matemáticas que, en lugar de escribirse como funciones, se insertan como símbolos entre los operandos (como + y -). Este compilador utiliza los mismos operadores del lenguaje C estándar, y con su misma precedencia y asociatividad.

Estos son todos los operadores disponibles, agrupados por tipos:

Operadores aritméticos

Operador	Significado
`+a`	a (ningún efecto)
`-a`	Cambio de signo
`a + b`	Suma
`a - b`	Diferencia
`a * b`	Producto
`a / b`	División
`a % b`	Módulo

Operadores de comparación

Operador	Significado
`a == b`	Igual a
`a != b`	Distinto de
`a < b`	Menor que
`a <= b`	Menor o igual que
`a > b`	Mayor que
`a >= b`	Mayor o igual que

Operadores lógicos

Operador	Significado
`!a`	NOT lógico
`a || b`	OR lógico
`a && b`	AND lógico

Operadores de bits

Operador	Significado
`~a`	NOT binario
`a | b`	OR binario
`a & b`	AND binario
`a ^ b`	XOR binario
`a << b`	Desplaza bits a izquierdas
`a >> b`	Desplaza bits a derechas

Operadores incremento y decremento

Operador	Significado
`a++`	a se evalúa y DESPUÉS se incrementa en 1
`a--`	a se evalúa y DESPUÉS se decrementa en 1
`++a`	a se incrementa en 1 y DESPUÉS se evalúa
`--a`	a se decrementa en 1 y DESPUÉS se evalúa

Operadores de asignación

Operador	Significado
`a = b`	Asignación
`a += b`	a = a + b
`a -= b`	a = a - b
`a *= b`	a = a * b
`a /= b`	a = a / b
`a %= b`	a = a % b
`a |= b`	a = a | b
`a &= b`	a = a & b
`a ^= b`	a = a ^ b
`a <<= b`	a = a << b
`a >>= b`	a = a >> b

Operadores para punteros

Operador	Significado
`&a`	Dirección de memoria de a (crea un puntero a a)
`*a`	Valor al que apunta a (contenido de esa dirección de memoria)
`!a`	a == NULL

Operadores para arrays

Operador	Significado
`a[ b ]`	Elemento con índice b dentro del array a

Operadores para estructuras y uniones

Operador	Significado
`a . b`	Miembro b dentro de a
`a -> b`	Miembro b dentro de la estructura a la que apunta a

Otros operadores

Operador	Significado
`(a)`	Las operaciones de dentro se agrupan y evalúan antes que las demás
`sizeof(a)`	Devuelve el tamaño en palabras de su argumento

Promoción de tipos

Como programadores debemos saber de que el lenguaje hace conversiones automáticas entre tipos de datos básicos cuando sea necesario. Por ejemplo, si hacemos esto:

```
int PlayerScore = 28.3;   // el float necesita ser convertido a int
```

El compilador convertirá automáticamente este valor en un entero y la variable almacenará realmente 28. Algo similar ocurre también cuando una expresión opera con distintos tipos: para sumar int + float, primero hay que convertir el int a float para que la operación sea válida entre tipos coherentes (float + float).

Llamadas a funciones

Podemos llamar a cualquier función que ya se haya declarado usando la notación estándar de paréntesis y comas:

```
// sumar una variable y una constante
int Sum = SumValues( Variable, 5 );

// las llamadas a funciones se pueden anidar
PrintNumber( SumValues( Variable, 5 ) );
```

Cuando una función no devuelve ningún valor (es decir, su tipo de retorno es void), una llamada a esa función es una expresión que no produce ningún resultado y, por lo tanto, no puede ser utilizada por otras expresiones.

Operador sizeof

El operador `sizeof()` determina el tamaño (siempre en palabras de 32 bits) de o bien un tipo de datos, o bien el resultado de una expresión.

```
// tamaño del resultado de una expresion
int Size1 = sizeof( 2+5 );        // Size1 = 1 (int)

// tamaño de un tipo de datos
int Size2 = sizeof( int[2][5] );  // Size2 = 10
```

Control de flujo

Un programa en C no se construye sólo con secuencias de expresiones, sino que también necesita herramientas para controlar el flujo de ejecución. Se dispone de las siguientes.

Condiciones

La forma más básica de controlar dónde continuará la ejecución es una simple condición. Utilizando `if` y `else` podemos evaluar una condición y determinar qué debe ocurrir si se cumple, y qué cuando no se cumple

También podemos encadenar varios ifs para comprobar condiciones relacionadas.

```
// ifs encadenados
if( x > 0 )
  print( "positive" );

else if( x < 0 )
  print( "negative" );

else
{
    print( "zero" );
    ShowAlert();
}
```

Switches

Si necesitamos elegir entre varios valores enteros, en vez de encadenar múltiples condiciones if-else también podemos usar `switch` para elegir entre una serie de casos:

```
switch( WeaponPowerLevel )
{
    // impacto de potencia media
    case 1:
    case 2:
        Player.Health -= WeaponPowerLevel;
        break;

    // impacto fuerte que destruye al jugador
    case 3:
        MakePlayerExplode();
        break;

    // para impactos muy debiles, o posibles casos de error
    default:
        MakeBulletBounce();
        break;
}
```

Bucles

C permite hacer saltos directos usando `goto`, pero los bucles son una alternativa mejor. Nos permiten repetir una sección del programa hasta que se cumpla una condición. En C hay 3 tipos de bucles: `while`, `do` y `for`.

```
// while comprueba la condicion al principio
while( x != 0 )
  x /= 2;

// do comprueba la condicion al final
do
{
    x -= 7;
    y = x;
}
while( x > 0 );

// for es un bucle mas configurable: incluye una instrucción
// de inicio, una condición de salida y una instrucción al iterar
for( int i = -5; i <= 5; i += 2 )
  print_number( i );
```

Control de bucles

Dentro de un bucle podemos usar `break` para terminar el bucle y continuar la ejecución donde éste termine. Por otro lado, el uso de `continue` nos permite avanzar a la siguiente repetición del bucle sin necesidad de llegar al final del mismo.

```
while( Enemy.EnemyID < MaxEnemyID )
{
    ++Enemy;

    // procesar solo enemigos activos
    if( !Enemy.Active )
      continue;

    ProcessEnemy( &Player, Enemy );

    // no hace falta continuar si el jugador ha muerto
    if( Player.Health <= 0 )
      break;
}
```

El preprocesador

El lenguaje C realiza un preprocesado de todo el texto de nuestros programas. El preprocesador recorre línea por línea los ficheros que componen el programa buscando directivas y aplica aquellas que pueda encontrar y reconocer.

Las directivas comienzan con el carácter almohadilla `#`, que debe ser el primer carácter de la línea (salvo espacios en blanco). Las directivas que se pueden utilizar son las siguientes.

Directiva #include

Usar `#include` nos permite to insertar en algún punto de nuestro código el contenido de otro archivo. Es útil porque nos permite organizar nuestro programa separando sus diferentes características en archivos independientes.

```
// incluir nuestras cabeceras
#include "Enemies\\Boss.h"
```

El preprocesador busca primero el archivo en la carpeta de la librería estándar y, después, en el directorio del archivo fuente. La ruta se trata como cualquier otra cadena de texto: si contiene caracteres especiales como '\', deben escribirse con su secuencia de escape.

Directivas #define y #undef

El preprocesador mantiene una lista de variables internas (no confundir con las variables del programa en C). Podemos definir una variable con `#define`, y borrar una definición con `#undef`. Las definiciones pueden tener un valor pero también pueden estar vacías.

```
// definicion simple con un valor literal
#define BallDiameter  16

// definicion que usa la anterior en una expresion
#define BallRadius  (BallDiameter / 2)

// ahora eliminamos otras definiciones anteriores
#undef BallSize
```

La directiva define tiene algunas limitaciones en este compilador. No puede utilizarse con parámetros como en el preprocesador estándar de C.

La librería estándar

Cualquier compilador de C debe implementar una serie de funciones para distintas materias (matemáticas, manejo de texto, etc.) para que los programas puedan acceder a unas funcionalidades mínimas, que funcionen de manera uniforme. Los programas pueden acceder a estas funciones incluyendo las cabeceras estándar del lenguaje.

La librería C estándar para Vircon32 incluye un buen número de funciones C estándar, pero no todas. Por otro lado, al tratarse de un compilador para una consola específica, la librería estándar de Vircon32 también añade funciones para trabajar con los diferentes sistemas de la consola: audio, vídeo, mandos, etc.

La colección actual de cabeceras del compilador de Vircon32 es la siguiente:

“audio.h”

Se usa para operar el chip de audio y reproducir sonidos.

“input.h”

Permite leer el estado de los mandos.

“math.h”

Las funciones matemáticas más comunes del lenguaje C.

“memcard.h”

Permite acceder a la tarjeta de memoria y leer o guardar datos.

“misc.h”

Funciones varias: gestión de memoria, números aleatorios y otras.

“string.h”

Incluye funciones para construir y manejar cadenas de texto.

“time.h”

Nos permite medir el paso del tiempo y controlar la velocidad de los programas.

“video.h”

Sirve para acceder al chip de video y mostrar imágenes en pantalla.