

Vircon32: Guide for the C compiler

Document date 2022.02.25

Written by Carra

What is this?

This document is a quick guide on how to use the C language to create programs in the Vircon32 console. The purpose of this guide is to show the features of the language, and its differences compared to the standard and other compilers. This document is applicable to compiler version 2022.02.25.

Summary

This document is organized into sections, each covering one of the features of the Vircon32 C language and its compiler.

| | |
|------------------------------------------|----|
| Summary | 1 |
| Introduction | 2 |
| Comments | 2 |
| Data types | 3 |
| Literal values | 5 |
| Variables | 6 |
| Functions | 8 |
| Expressions | 9 |
| Flow control | 10 |
| Embedding assembly code | 13 |
| The preprocessor | 13 |
| The standard library | 15 |
| Global differences with standard C | 16 |
| General compiler limitations | 17 |
| Using the compiler | 18 |

Introduction

The C compiler is part of the Vircon32 development tools, and its function is to interpret our programs written in C language to translate them into assembly language. Then, a second program (the assembler) will in turn translate that into machine code instructions that the Vircon32 CPU is able to execute.



The C language used in this compiler is mostly the same as the standard, but there are some differences and limitations that need to be taken into account when programming. Some of these differences are given by the way the console itself works, and others are simply the result of my own limitations when programming a compiler. Still I think it is much more valuable to be able to program at a higher level, even if it comes with some limitations, than to place barriers to the creation of games on the console by requiring to learn assembler.

In the following sections I will describe the features of the language, and **highlight in red** the differences with a standard C compiler, so that those who already know how to program in C language can see at a quick glance the main things they should watch for and take into account.

Comments

Comments are a way to include in the program our own explanations without having the compiler try to interpret them (which would cause errors). That is to say, comments are not part of the program itself. There are 2 types of comments, and they work the same way as in the standard C language:

Line comment

These comments begin with `//`, and finish at the end of the line. They can be extended to the following lines if the line is continued by ending it with `\`.

```
// line comment that is continued \  
in the next line  
  
int x = 7 + 5;    // comment not continued  
int y = x;
```

Block comment

A block comment begins with `/*`, and will not finish until we end it with `*/`, even if it's done in a different line.

```
/* this comment negates the line below
int x = 7 + 5;
*/

int y;
```

Data types

All values or variables handled by the language belong to some data type that indicates how to interpret that value. Here we will see what data types we can use.

Basic types

The Vircon32 C language, unlike standard C, has only data types of size 32 bits. Moreover, there are no variants of these types: neither for sign (signed, unsigned), nor for size (short, long).

Vircon32 organizes memory in 32-bit words because in this console the minimum unit of memory is the word (and not the byte), so we must take into account that:

- All sizes for data types are measured in words and not bytes
- All memory addresses and offsets are given in words and not bytes

The 4 basic data types in this compiler are:

| | | | |
|-----|-------|------|------|
| int | float | bool | void |
|-----|-------|------|------|

All of these types have size 1 (i.e. 32 bits). Type void is special, and can only be used to indicate that a function does not return any value.

Derived types

We can create new types using the basic types in arrays and pointers. This can be done multiple times, for example:

| | | | |
|------|--------|-------------|----------|
| int* | void** | float[3][5] | void*[4] |
|------|--------|-------------|----------|

We can use void to create pointers, but not arrays (because there can be no void type values). A pointer to void is used to store a memory address where information is read or stored without interpreting it (void is used to indicate the absence of type).

Function pointers

In the current compiler version there are no function pointers. However, they are planned as a feature to be added in next versions.

Compound types

We can group several data types into a larger type, by creating structures and unions. Each member of these groups is a field that is accessed by name.

Whereas in standard C declaring a struct/enum declares a variable and not a type (unless typedef is used), the C language used in Vircon32 treats these as type declarations, in a way more similar to how C++ treats a class declaration:

```
// declare type Point
struct Point
{
    int x, y;
};

// declare type Word
union Word
{
    int AsInteger;
    void* AsPointer;
};

// use some structures
Point P1,P2;
P1.x = 10;
P1.y = -7;
P2 = P1;

// use some unions
Word W;
W.AsInteger = 0xFF110AF;
int* Pointer = W.AsPointer;
```

Structures and unions, in turn, may also contain other structures and unions. They can also include themselves, but only through pointers for obvious reasons.

Standard C allows structures to use bit fields in order to group individual bits. They are not supported here, and there are no plans to add them.

Enumerations

We can define a series of integer constants and group them into their own type. The C language allows this through enumerations. In general these values are treated as integers and you can use them in the same ways, but their type is more restrictive. Same as with struct and union, using enum will always declare a type as if using typedef.

If no values are specified, first constant will have a numerical value of 0 and every following constant has a value of the previous one + 1.

```
// declare type Semaphore
enum Semaphore
{
    Red = 1,
    Yellow,
    Green,
};

// these operations are fine
Semaphore S1,S2;
S1 = Red;
S2 = S1;
int Value = Yellow + Green; // can safely convert enum to int

// these assignments would produce errors!
S1 = Green - Red;
S2 = 1; // same value as Red, but can't convert int to enum
```

Defining types with typedef

We can use `typedef` to give a name to a data type. That name can then be used to represent said data type.

```
// we define a type
typedef int** int_ptr2;

// and then we use it
int_ptr2 P = NULL;
```

Remember that **in this compiler, the way of writing complex types is simplified compared to standard C**. For example: in regular C, to define a pointer to an array, it is necessary to use parentheses and place the name inside of the type itself:

```
typedef int (*ptr_to_array)[5];
```

Instead, in this compiler they are always kept separate: <type> <name>. This makes types more easy to identify, by reading from right to left:

```
typedef int[5]* ptr_to_array;
```

Literal values

In our programs we can use constant values that we write literally. Depending on the data type each of these values is representing, we have different notations and numerical representations. In this compiler there exist the following:

```
-15;    // int in decimal
0xFF1A; // int in hexadecimal
0.514;  // float
true;   // bool
'a';    // int as a character (char does not exist)
"hola"; // int[5] (string of 4 characters + terminating 0)
NULL;   // null pointer
```

The values for boolean literals are: true = 1, false = 0.

Note that, while in most C compilers the value of NULL is 0, in this compiler its value is actually -1. It was chosen that way because in Vircon32 memory address 0 is a valid address.

Even then, operations with NULL can still be done as they usually would. For example, here a check such as `if(pointer)` will test if the pointer is different from -1.

For float values scientific notation (for instance: 1.35e-7) is currently not supported.

Variables

In addition to constant data we can also handle variables: they are memory addresses where the value that varies is stored. This value is accessed by using a name that identifies the variable, and a data type that interprets the stored value.

Declaring variables

A variable is declared with a type and a name, and can optionally be initialized with a value:

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

As we saw when talking about typedef, types in this compiler are always kept separate from the name, in this case: <type> <name> (= <value>). For example, to declare an array in this compiler it must be done like this:

```
// this is an arrays that has 20 arrays, each of which has 10 ints
int[20][10] LevelBricks;
```

If we try to declare an array as in standard C, the compiler will raise an error.

Multiple declarations

It is also possible to declare several variables of the same type in a single declaration, separating them with commas:

```
// declare 3 pointers to int
int* ptr1 = &number, ptr2 = ptr1, ptr3;
```

Note that in this compiler all variables declared at the same time are always of the same type (here, they are all `int*`), while in standard C only the first of them would be a pointer.

Initialization lists

Arrays and structures can be initialized with a list of multiple values. These lists can also be nested for more complex types, as seen in these examples:

```
// initialize a structure
struct Point
{
    int x, y;
};

Point P = { 3, -7 };

// use nested lists for an array of structures
Point[3] TrianglePoints = { {0,0}, {1,0}, {1,1} };

// for int arrays we can also use a string instead of a list
int[10] Text = "Hello"; // careful, C add 1 extra character (0) as termination
```

Kinds of variables

In C language 2 basic kinds of variables exist: locals and globals.

- A local variable is declared in the body of a function, and is only accessible within its scope (because it's stored in the stack, which changes through execution).
- A global variable is declared outside functions and is accessible to the whole program after its declaration because its address is fixed.

In Vircon32 C, a third type of variable is added: embedded files. They are declared using `embedded` as in the following example:

```
// the contents of this variable are read from the file during compilation
embedded int[200][100] TileMap = "GameData\\LevelMap.bin";
```

What the compiler will do in this case is to store that array in the cartridge instead of in RAM memory. This allows us to use large volumes of data without having to write it in the code, although being in the cartridge that data is always read-only. This mechanism allows us to use external data in our program, even though there are no files from which to load them externally as is done on a computer.

It is important to know that the compiler requires that this file, besides existing, has the same size as the variable that stores it. That is to say, in this case the size of file LevelMap.bin file must be equal to:

Number of ints = $200 \times 100 = 20,000$
Bytes per int = 4 (these are 32-bit ints)
Total size = 80,000 bytes

Embedded files can only be used at the global scope, this is, they cannot appear inside functions (since they are never stored in the stack).

Modifiers

In this compiler, for the moment, there are no modifiers to apply in variable declarations: **const, static, volatile, and register are not supported.** The only one of them that could possibly be implemented in the future is const.

Functions

In C language it is now allowed to execute any statements outside of a function beyond just declarations. In order to execute code in our program, we will need to declare functions that contain the statements to be executed. Functions in Vircon32 C are declared in the same way as in standard C (although they use this compiler's data type notation as we already saw), for example:

```
// function that adds 2 integers
int AddValues( int a, int b )
{
    return a + b;
}
```

The body of the function can contain multiple statements that will be executed in a sequential manner.

An important difference with standard C is that, due to compiler limitations, **functions cannot use parameters or return values of size different from 1. That is: they cannot use arrays, unions or structures (unless their size is just a single word).** Instead they must operate with pointers to them.

It is however possible to "pass an array" as a parameter when the array decays to a pointer, in the same way as in standard C language. That is, we may do this:

```
// function that adds N integers
int SumValues( int* Values, int N )
{ // ... }
```



```
// array of 10 integers
int[10] Values;

// we add the first 5 values in the array
int Sum = SumValues( Values, 5 );
```

In C language it is allowed to use ellipses in the argument list of a function so that it can receive an indeterminate number of parameters. This is used in standard functions such as printf. **That mechanism is not supported in this compiler.**

Expressions

Expressions are operations that we can do by combining variables and literal values. These combinations can be made by applying functions or operators to them, until we obtain a single final value as a result.

Function calls

We can call any function that has already been declared using the standard notation of parentheses and commas:

```
// add a variable and a constant
int Sum = SumValues( Variable, 5 );

// function calls can be nested
PrintNumber( SumValues( Variable, 5 ) );
```

When a function does not return any value (i.e. its return type is void), a call to that function is an expression that does not produce a result and therefore cannot be used by other expressions.

Operators

For the most part operators represent mathematical operations that, instead of being written as functions, are inserted as symbols between the operands (such as + and -). This compiler uses the same operators of the standard C language, and with their same precedence and associativity.

The only exceptions are the ternary operator `a ? b : c` and the comma operator `a , b`, which are not supported in this compiler.

Special operators

The `sizeof()` operator determines the size (always in 32-bit words) of either a data type or an expression's result.

```
// size of an expression's result
int Size1 = sizeof( 2+5 );      // Size1 = 1 (int)

// size of a data type
int Size2 = sizeof( int[2][5] ); // Size2 = 10
```

In standard C language there is also an explicit type conversion operator. **It does not exist in the current version of the compiler, but it is planned to incorporate it in the future.** However, the compiler still makes implicit conversions where needed.

Flow control

A C program is not only built with sequences of expressions, but will also need tools to control the execution flow. The following are available.

Conditions

The most basic way to control where execution will continue is a simple condition. Using `if` and `else` we can evaluate a condition and determine what should happen if it is met, and what when it is not met.

We can also chain together several ifs to check for related conditions.

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

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

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

Switches

If we need to choose from several integer values, instead of chaining multiple if-else conditions we can also use `switch` to select from a series of cases:

```
switch( WeaponPowerLevel )
{
    // mid-power impact
    case 1:
    case 2:
        Player.Health -= WeaponPowerLevel;
        break;
```

```

    // hard impact that destroys player
    case 3:
        MakePlayerExplode();
        break;

    // for impacts too weak, or possible error cases
    default:
        MakeBulletBounce();
        break;
}

```

Labels and goto

We can define labels inside a function to mark certain positions in the code. Then we can use `goto` to make our program execution jump to those positions.

```

// labels are only valid inside functions
Void ProcessData( DataStructure* Data )
{
    Step1:

    // avoid this step in some cases
    if( SkipPreprocessing )
        goto Step2;

    Preprocess( Data );

    Step2:

    // keep processing data until errors are low enough
    if( !Transform( Data ) )
        goto Step1;
}

```

Loops

Loops are a better alternative than goto. They allow us to repeat a section of the program until a condition is met. There are 3 types of loops in C: `while`, `do` and `for`.

```

// while checks the condition at the beginning
while( x != 0 )
    x /= 2;

// do checks the condition at the end
do
{
    x -= 7;
    y = x;
}
while( x > 0 );

// for is a more configurable loop
for( int i = -5; i <= 5; i += 2 )
    print_number( i );

```

Loop control

Inside a loop we can use `break` to terminate the loop and continue execution after its end. On the other hand, using `continue` allows us to advance to the next repetition of the loop without having to reach the end of the loop.

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

    // process only active enemies
    if( !Enemy.Active )
        continue;

    ProcessEnemy( &Player, Enemy );

    // no need to continue if player has died
    if( Player.Health <= 0 )
        break;
}
```

Returning from functions

Inside the body of a function, we can use `return` to exit the function at any time. The execution will return to the point in the program where that function was called.

Return is also used to return a value (when the return type of the function is not void). In that case return must be used with a value of compatible type.

```
// function that does not return a value
void DoNothing()
{
    return;
}

// function that returns a pointer
int* FindLetterA( int* Text )
{
    while( Text )
    {
        if( *Text == 'A' )
            return Text;

        Text++;
    }

    return NULL;
}
```

Embedding assembly code

Vircon32 C language, like some other implementations, allows you to embed assembly language instructions using `asm`. This feature can only be used inside the body of a function. The syntax for this is very similar to the one used by the gcc compiler, and a text string is used to write each instruction.

```
// a C function implemented in assembly
void select_sound( int sound_id )
{
    asm
    {
        // here we use the value for parameter sound_id
        "mov R0, {sound_id}"
        "out SPU_SelectedSound, R0"
    }
}
```

We can also use the variables of our C program in the assembler instructions. To do this we write the variable name between braces.

This way of connecting C and assembler has some limitations: it only supports names, and not expressions. For example, we could not use it to read a member of an array. Also, only one variable is allowed to be referenced in each instruction. Because of the way Vircon32 CPU works, it will never be possible to use more than one variable at a time.

The preprocessor

C language performs a preprocessing of all text in our C programs. The preprocessor walks line by line through the files that make up the program looking for directives and applies those it can find and recognize.

Directives begin with the hash character `#`, which must be the first character in the line (except for whitespace). The directives that can be used are the following.

Directive `#include`

Using `#include` allows us to insert at some point in our code the content of another file. It is useful because it enables us to organize our program by separating its different features in separate files.

```
// include our headers
#include "Enemies\\Boss.h"
```

The preprocessor will first look for the file in the standard library folder, and then in the directory of the source file itself.

In C it is allowed to write the file path between quotes ("path") or between angles (<path>). This compiler only allows us to use quotes. Also, the path is treated like any other text string: if it contains special characters such as '\', they must be written using their escape sequence.

Directives #define and #undef

The preprocessor maintains a list of internal variables (not to be confused with C program variables). We can define a variable with `#define`, and delete a definition with `#undef`. Definitions can have a value but they can also be empty.

```
// simple definition with a literal value
#define BallDiameter 16

// definition that uses the previous one in an expression
#define BallRadius (BallDiameter / 2)

// now remove other old definitions
#undef BallSize
```

The define directive has some limitations in this compiler. It cannot be used with parameters as in the standard C preprocessor. In addition, the use of the `#` and `##` operators within definitions to form strings is not supported.

Conditional directives

In C it is possible to use the preprocessor to delimit parts of the program that should be compiled and others that should not. Such conditional compilation is achieved by using directives that check whether a preprocessor variable is defined or not, and its value. The conditional directives that allow this are `#ifdef`, `#ifndef`, `#else` and `#endif`.

```
#define DEBUG // commenting this line would change the program

#ifdef DEBUG
    Debug = true;
#else
    Debug = false;
#endif
```

One limitation to note is that in this compiler, not compiled parts are not parsed but they are read at a basic level (to recognize strings, numbers, etc). Therefore we might receive some errors even in regions that are not compiled.

Directives `#if` and `#elif` from standard C language are not supported at the moment.

Directives #error and #warning

These directives allow us to force the compiler to display messages during compilation. If we use `#warning`, the compiler will only display a warning. On the other hand, with `#error` compilation will be stopped.

```
// after this line our compilation still continues
#warning "careful, error on the next line"

// but this line will halt it with an error
#error "this part should not have been compiled"
```

Other non supported features

Other directives from C language, like `#line` and `#assert`, are not supported.

Additionally, in a standard C preprocessor there are some internal variables which are `__FILE__`, `__LINE__`, `__DATE__`, `__TIME__` and `__func__`. They are used to create text strings dynamically at compile time, depending on different parameters: current time, the name of the source file, etc. **In this compiler these variables do not exist.**

The standard library

Any C compiler must implement a series of functions for different subjects (mathematics, string handling, etc.) so that programs can access some minimum functionality, that works in a uniform manner. These functions are accessible to programs by including the standard language headers in the code.

The standard C library for Vircon32 includes a good number of standard C functions, **but not all of them. One of the most notable absences as of now are the functions for using dynamic memory (malloc, calloc, realloc and free).**

On the other hand, since this is a compiler for a specific console, **the Vircon32 standard library also adds functions to work with the different console systems: audio, video, gamepads, etc.**

The current collection of headers in the Vircon32 compiler is the following:

“audio.h”

It is used to operate the audio chip and play sounds.

“input.h”

It allows to read the state of gamepads.

“math.h”

The most common mathematical functions of the C language.

“memcard.h”

Used to access the memory card and read or save data.

“misc.h”

Miscellaneous functions: memory management, random numbers and others.

“string.h”

Includes functions for building and handling text strings.

“time.h”

It allows us to measure the flow of time and control the speed of the programs.

“video.h”

Used to access the video chip and show images on screen.

Global differences with standard C

The main function

The main function is a special function that must always exist, since execution of our code always starts at the beginning of this function. In the standard C language, the prototype for the main function is as follows:

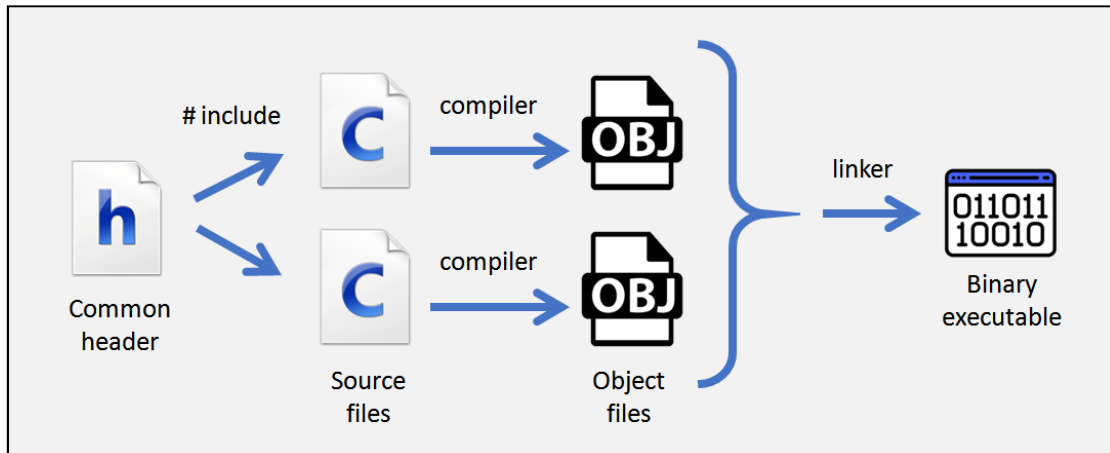
```
int main( int NumberOfArguments, char* Arguments[] )
{
    // ...
}
```

Vircon32 is a console in which there is no operating system and, after the initial BIOS startup, our program is the only one running on the machine. Since there is no other program that can pass parameters to us or to which we can return a value, **in Vircon32 it makes no sense to have that prototype for main. Therefore, the main function in this compiler must be declared as follows:**

```
void main()    // either this or: void main( void )
{
    // ...
}
```


Structure of a program

A standard C compiler allows you to split a program into several "compilation units". This means that you can have several files with their own variables and functions that are compiled separately and then linked together using a linker.



Normally, each of these files can access each other's functions by including a common header that defines the prototypes of those functions, but does not implement them. This is called a partial declaration.

This compiler follows a more simplified structure: **there are no partial declarations, and there is no linker that can join several compiled files. The way a source file can use functions from other files is to include those files from the main file, with all their complete code.**

The standard library, for example, is also made in this way and includes the complete implementation for all its functions.

General compiler limitations

The Vircon32 C compiler has been programmed by a single person having limited knowledge on how compilers are built. Although it works relatively well, it has some limitations compared to what a more serious compiler would be in terms of the compilation process itself.

Optimization

This compiler does not produce very optimized code. It will only detect certain very simple situations in which to optimize, and if we analyze the assembly code it generates, it will not be hard to find sections that have unnecessary instructions.

Error detection

Error detection in the compiler correctly reports the location of each error in the code. However in many situations it will stop the program at the first error detected, while other compilers may give several errors at the same time.

Character set

At the moment this compiler does not support Unicode. Some non-English characters can still be used, but the source files will be interpreted following Windows Latin 1/CodePage 1252 encoding. The text font included in the BIOS (the one used by print functions in standard library) is also encoded in this way. This encoding includes 256 characters, the first 128 being the standard ASCII characters. The rest include all special characters from Spanish, Portuguese, French and German, as well as some commonly used symbols such as the Euro.

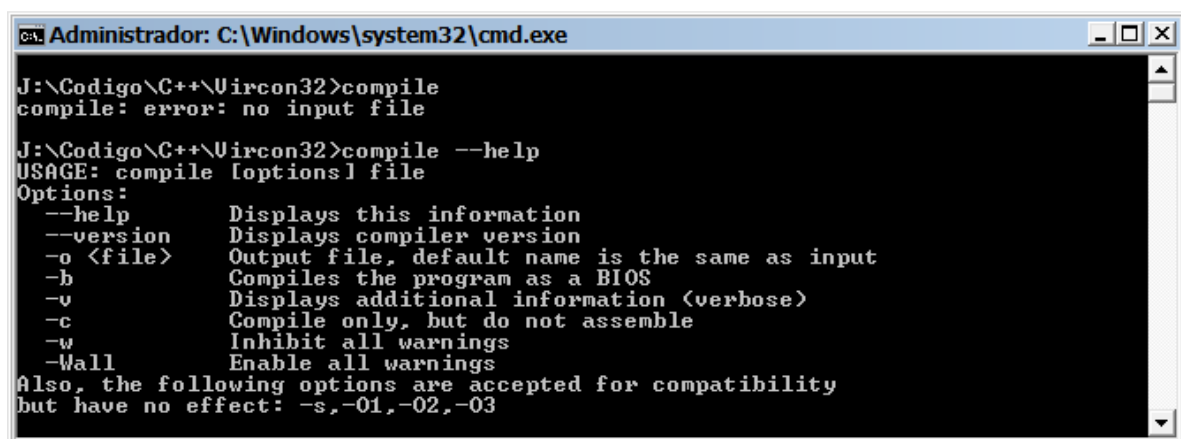
Using the compiler

Broadly speaking there are 2 ways to use the compiler, and the set of Vircon32 development tools in general.

Command line

Vircon32 tools are command line programs, so the program can either be started by hand or called with a script to compile as needed. The code can be written in any text editor. The disadvantage of using the command line is that whenever the compiler informs us of any error we will have to read the message and locate the error in our code manually.

The input parameters for the compiler have been chosen to be used in the same way as the gcc compiler, as shown in the image below. That is, the command structure is: `compile.exe [options] file`.



```
Administrador: C:\Windows\system32\cmd.exe

J:\Codigo\C++\Vircon32>compile
compile: error: no input file

J:\Codigo\C++\Vircon32>compile --help
USAGE: compile [options] file
Options:
--help          Displays this information
--version       Displays compiler version
-o <file>       Output file, default name is the same as input
-b             Compiles the program as a BIOS
-v             Displays additional information <verbose>
-c             Compile only, but do not assemble
-w             Inhibit all warnings
-Wall          Enable all warnings
Also, the following options are accepted for compatibility
but have no effect: -s,-01,-02,-03
```

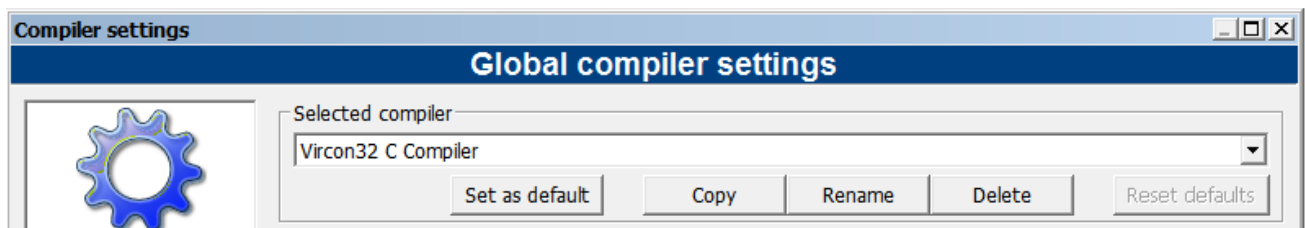
Even using the same parameters, there are still 2 important differences in its use when compared to gcc:

- This compiler only generates assembly code (with default extension .asm), and will not create binaries by itself. To do this, it is needed to call the assembler program `assemble.exe` afterwards.
- The binary file that is created with the assembler is not directly executable by Vircon32 either. This is because we must package the binary in a ROM using the packer program `packrom.exe`. In this way the ROM integrates the binary together with its images and sounds, if they are used.

Integration in an IDE

The display format of Vircon32 compiler errors and warnings has also been chosen to match those of the gcc compiler. This will make it easier to integrate the compiler into development environments if they are sufficiently configurable.

For example, one of the options would be to create a configuration for the Vircon32 compiler starting from the gcc one (which should exist in any IDE), and configure our projects to use it.



Specifically, in the environment in Code::Blocks you can work with Vircon32 tools in several ways. In fact, by scripting you can even add custom option menus as seen in the image below.

