# C64 Language Reference

Table of Contents

[C64 Language Reference 1](#_Toc489469381)

[Overview 2](#_Toc489469382)

[Compiler Options 3](#_Toc489469383)

[\_\_check 5](#_Toc489469384)

[align() 6](#_Toc489469385)

[and 6](#_Toc489469386)

[asm [\_\_leafs] 6](#_Toc489469387)

[case 8](#_Toc489469388)

[class 8](#_Toc489469389)

[epilog 9](#_Toc489469390)

[firstcall 10](#_Toc489469391)

[forever 10](#_Toc489469392)

[inline 10](#_Toc489469393)

[if 10](#_Toc489469394)

[nocall / naked 11](#_Toc489469395)

[or 11](#_Toc489469396)

[pascal 11](#_Toc489469397)

[prolog 11](#_Toc489469398)

[switch 13](#_Toc489469399)

[then 13](#_Toc489469400)

[thread 14](#_Toc489469401)

[typenum() 14](#_Toc489469402)

[until 14](#_Toc489469403)

[using 15](#_Toc489469404)

[Block Naming 16](#_Toc489469405)

[Array Handling Differences from ‘C’ 17](#_Toc489469406)

# Overview

C64 supports an extended ‘C’ language compiler. C64 is able to compile most C language programs with little or no modification required. In addition to the standard ‘C’ language C64 adds the following:

run-time type identification (via typenum())

exception handling (via try/throw/catch)

function prolog / epilog control

multiple case constants eg. case ‘1’,’2’,’3’:

assembler code (asm)

pascal calling conventions (pascal)

no calling conventions (nocall / naked)

inline code

additional loop constructs (until, loop, forever)

true/false are defined as 1 and 0 respectively

thread storage class

structure alignment control

firstcall blocks

block naming

branch prediction hints

# Compiler Options

|  |  |
| --- | --- |
| Option | Description |
| -fno-exceptions | This option tells the compiler not to generate code for processing exceptions. It results in smaller code, however the try/catch mechanism will no longer work. |
| -o[pxrc] | This option disables optimizations done by the compiler causing really poor code to be generated.  p – this disables the peephole optimization step  x – this disables optimization of expressions (constants)  r – this disables the allocation of register variables and common subexpression elimination  c – this disables optimizations done during code generation  -o by itself disables all optimizations done by the compiler |
| -w | This option disables wchar\_t as a keyword. This keyword is sometimes #defined rather than being built into some compilers. |
| -S | generate assembly code with source code in comments. |

OPC5 Register Usage

|  |  |  |  |
| --- | --- | --- | --- |
| Reg |  |  | Saved by |
| r0 | always zero | fixed use by hardware | --- |
| r1 | return value / temporary | by convention | caller |
| r2 | register variable |  | callee |
| r3 | register variable |  | callee |
| r4 | register variable |  | callee |
| r5 | temporary | expression processing | caller |
| r6 | temporary | expression processing | caller |
| r7 | temporary | expression processing | caller |
| r8 | parameter | register parameter to functions | caller |
| r9 | parameter | register parameter to functions | caller |
| r10 | parameter | register parameter to functions | caller |
| r11 | class pointer | a pointer to the class data structure | caller |
| r12 | base pointer |  | callee |
| r13 | link register | stores the return address | caller |
| r14 | stack pointer | by convention | callee |
| r15 | program counter | fixed use by hardware | --- |

The following additions have been made:

typenum(<type>)

allow run-time type identification. It returns a hash code for the type specified. It works the same way the sizeof() operator works, but it returns a code for the type, rather than the types size.

C64 supports a simple try/throw/catch mechanism. A catch statement without a variable declaration catches all exceptions.

try { <statement> }

catch(var decl) {

}

catch(var decl)

{

}

catch {

}

Types:

A byte is one byte (8 bits) in size.

A char is two bytes (16 bits) in size.

An int is eight bytes (64 bits) wide.

An short int is four bytes (32 bits) wide

Pointers are eight bytes (64 bits) wide.

# \_\_check

\_\_check causes the compiler to output a bounds checking instruction. The bounds expression must be of the format shown in the example.

Example:

\_\_check (hMbx >= 0 && hMbx < 1024);

# ****align()****

The align keyword is used to specify structure alignment in memory. For example the following structure will be aligned on 64 byte boundaries even though the structure itself is smaller in size.

struct my\_struct align(64) {

byte name[40];

}

Place the align keyword just before the opening brace of a structure or union declaration.

Note that specifying the structure alignment overrides the compiler’s capability to automatically determine structure alignment. Care must be taken to specify a structure alignment that is at least the size of the structure.

Taking the size of a structure with an alignment specified returns the alignment.

# and

‘and’ is defined as a keyword and is a synonym for ‘&&’. It can make code a little more readable.

Example:

if (a and b) {

}

# **asm [\_\_leafs]**

The asm keyboard allows assembler code to be placed in a ‘C’ function. The compiler does not process the block of assembler code, It simply copies it verbatim to the output. Global variables may be referenced by name by following the compiler convention of adding an ‘\_’ to the name. Stack parameters have to be specifically addressed referenced to the bp register.

pascal void SetRunningTCB(hTCB ht)

{

asm {

lw tr,24[bp] ; this references the ht variable

asli tr,tr,#10

addui tr,tr,#tcbs\_ ; this is a global variable reference

}

}

The \_\_leafs keyword indicates that the assembler code contains leafs (calls to other functions). Using the \_\_leafs keyword causes the compiler to emit code to save and restore the subroutine linkage register.

// ----------------------------------------------------------------------------

// Set an IRQ vector

// ----------------------------------------------------------------------------

pascal void set\_vector(unsigned int vecno, unsigned int rout)

{

if (vecno > 255) return;

if (rout == 0) return;

asm **\_\_leafs** {

lw r2,32[bp]

lw r1,40[bp]

jsr set\_vector

}

}

# case

Case statement may have more than one case constant specified by separating the constants with commas.

C64:

switch (option) {

case 1,2,3,4:

printf(“option 1-4);

case 5:

printf(“option 5”);

}

Standard C:

switch (option) {

case 1:

case 2:

case 3:

case 4:

printf(“option 1-4);

case 5:

printf(“option 5”);

}

# class

C64 features a simple class keyword which may be used to implement classes. A class is very similar to a struct except that class methods may be declared to be part of the class. Classes in C64 can have only single inheritance.

# ****epilog****

The epilog keyword identifies a block of code to be executed as the function epilog code. An epilog block maybe placed anywhere in a function, but the compiler will output it at the function’s return point.

nocall myfunction()

{

// other code

epilog asm {

// do some epilog work here, eg. setup return values

}

}

# firstcall

The firstcall keyword defines a statement that is to be executed only once the first time a function is called.

firstcall {

printf(“this prints the first time.”);

}

The compiler automatically generates a static variable in the data segment that controls the firstcall block. The firstcall statement is equivalent to:

static char first=1;

if (first) {

first = 0;

<other statements>

}

# forever

Forever is a loop construct that allows writing an unconditional loop.

forever {

printf(“this prints forever.”);

}

# inline

The inline keyword may be applied to a function declaration to cause the compiler to emit the function “inline” with other code. Every time the inline function is called, the code for the function is replicated inline.

# if

If statements can accept a branch hint to indicate if the branch should be statically predicted as taken (1) or not taken (0). The prediction must be a constant value determined at compile time. The syntax adds an options second expression ‘;’ into the expression clause as shown below.

if (a < 10; 1) // predict taken all the time

…

# nocall / naked

The nocall or naked keyword causes the compiler to omit all the conventional stack operations required to call a function. (Omits function prologue and epilogue code) It’s use is primarily to allow inline assembler code to handle function calling conventions instead of allowing the compiler to handle the calling convention. The naked keyword may also be applied to the switch() statement to cause the compiler to omit bounds checking on the switch.

nocall myfunction()

{

asm {

}

}

# or

‘or’ is defined as a keyword and is a synonym for ‘||’. It can make code a little more readable.

Example:

if (a or b) {

}

# pascal

The pascal keyword causes the compiler to use the pascal calling convention rather than the usual C calling convention. For the pascal calling convention, function arguments are popped off the stack by the called routine. This may allow slightly faster and smaller code in some circumstances.

pascal char myfunction(int arg1, int arg2)

{

}

# prolog

The prolog keyword identifies a block of code to be executed as the function prolog. A prolog block may be placed anywhere in a function, but the compiler will output it at the function’s entry point.

nocall myfunction()

{

prolog asm {

// do some prolog work here, eg. setup stack parameters

}

}

# switch

The naked keyword may be applied to the switch() statement to cause the compiler to omit bounds checking. Normally the compiler will check the switch variable to ensure that it’s within the range of the defined case values. With a naked switch the compiler assumes that the switch value is between the minimum and maximum case value in the switch statement. Naked switches result in faster code, but results are undefined if the switch is out of range. For a naked switch if the switch value isn’t valid then the program will likely crash. So use with caution.

Regular switch:

|  |
| --- |
| ; switch(btn) {  lw r3,-32[bp]  ldi r4,#1  ldi r5,#9  chk r3,r4,r5,BIOSMain\_13  sub r3,r3,#1  shl r3,r3,#3  lw r3,BIOSMain\_19[r3]  jal r0,0[r3] |

Naked Switch:

|  |
| --- |
| ; naked switch(btn) {  lw r3,-32[bp]  sub r3,r3,#1  shl r3,r3,#3  lw r3,BIOSMain\_19[r3]  jal r0,0[r3] |

Note that if the minimum case value is zero then the code may omit the subtract of the minimum value making the switch slightly faster.

# then

‘Then’ is defined as a keyword. It’s only purpose is to make code more readable. It may be used with ‘if’ statements in which case it is ignored.

# ****thread****

The ‘thread’ keyword may be applied in variable declarations to indicate that a variable is thread-local. Thread local variables are treated like static declarations by the compiler, except that the variable’s storage is allocated in the thread-local-storage segment (tls).

thread int varname;

# typenum()

Typenum() works like the sizeof() operator, but it returns a hashcode representing the type, rather than the size of the type. Typenum() can be used to identify types at run-time.

struct tag { int i; };

main()

{

int n;

n = typenum(struct tag);

}

The compiler numbers the types it encounters in a program, up to 10,000 types are supported. Pointers to types add 10,000 to the hash number for each level of pointer.

# until

Until is a loop construct that allows writing a loop that continues until a condition is true. Until and while are almost the same except that until waits for the inverted condition.

x = 0;

until (x==10) {

printf(“this prints 10 times.”);

x = x + 1;

}

# using

The using keyword is used to activate features of C64. In particular a name mangler used for classes can be activated by using the phrase ‘using name mangler;’ Without activating the name mangler all class methods have global scope.

# ****Block Naming****

The compiler supports named compound statement blocks. To name a compound statement follow the opening brace with a colon then the name.

void SomeFunc()

{

while (x) {: x\_name

<other statements>

}

}

An eventual goal for the compiler is to have the break statement be able to identify which block statement to break out of.

# Array Handling Differences from ‘C’

The following is “in the works”. It may or may not work.

Arrays may be passed by value using the standard declaration of an array as a parameter. In ‘C’ arrays are always passed by reference.

In C64:

SomeFn(int ary[50]) {

}

Declares a function that accepts an array of 50 integers passed by value. Declaring the function the same way in ‘C’ results in a reference to the array being passed to the function rather than the array values.

In order to pass an array by reference in C64 the pointer indicator ‘\*’ must be used as in the following:

SomeFn(int \*ary) {

}

It is not recommended to pass large arrays or structures around in a program by value as program performance may be adversely affected. Passing aggregate types by value causes the compiler to output code to copy the values .The alternative, passing references around is significantly faster.