CC64 Language Reference

Table of Contents

Overview

CC64 supports an extended ‘C’ language compiler. CC64 is able to compile most C language programs with little or no modification required. In addition to the standard ‘C’ language CC64 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. Also turns off constant optimizations. 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.

FT64 Register Usage

Register	Description / Suggested Usage	Saver
r0	always reads as zero	
r1-r2	return values / exception	caller
r3-r10	temporaries	caller
r11-r17	register variables	callee
r18-r22	function arguments	caller
r23	assembler usage	
r24	type number / function argument	caller
r25	class pointer / function argument	caller
r26	thread pointer	callee
r27	global pointer	
r28	exception link register	caller
r29	return address / link register	caller
r30	base / frame pointer	callee
r31	stack pointer (hardware)	callee

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.

CC64 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.

__attribute__

__attribute__ defines attributes associated with functions. Currently the only defined attribute is __no_temps which indicates to the compiler that the function does not use any temporary registers. This allows the compiler to omit code to save and restore temporaries around function calls. This is used primarily for functions defined in assembly language.

```
Example:
       extern signed byte KeybdGetStatus() __attribute__(__no_temps);
       extern byte KeybdGetScancode() __attribute__(__no_temps);
```

__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; 1024);
```

The first expression must be greater than or equal to the second expression and less than the third expression or a processor check interrupt will be invoked. Note any valid expressions may be used.

__mulf

__mulf causes the compiler to output a fast, single cycle multiply instruction. The fast multiply instruction is limited to 24 x 16 bits.

```
Example:
       ndx = __mulf (row, 56);
```

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 arguments have to be specifically addressed referenced to the fp register. Register arguments can use the register directly.

```
pascal void SetRunningTCB(hTCB ht)
{
   asm {
      lw     tr,32[fp]     ; this references the ht variable
      asli   tr,tr,#10
      add    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[fp]
```

```
                        lw              r1,40[fp]
                        call            set_vector
        }
}
```

case

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

CC64:

```
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”);
}
```

catch (<type>)

The catch statement “catches” a specific type of object used for exception handling. A catch handler corresponds to object type used in the throw statement. If the thrown object is not of a type caught by a local catch handler, then a search for the correct catch handler will continue at a more outer level.

class

CC64 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 CC64 can have only single inheritance.

delete (not working yet)

delete calls the run-time function __delete() to delete an object allocated by the new operator. __delete() takes a pointer to the object and a pointer to the function's object list and removes the object from the object list. It then also deallocates the object from the heap. If an object is deleted it is immediately deallocated and is not garbage collected. Delete does not call an object destructor. The object should be destroyed before using delete.

enum

The stride may be specified for the enumeration by following the enum keyword with the parenthesized stride value. The value must be a constant. If not specified the enumeration will increment by one.

In the following example the enumeration will decrement by 1 the value for each enumerated constant. so BADARG is equal to -1. This may be useful in cases where functions return a negative value indicating error or a positive value indicating proper operation.

```
enum(-1) {
        OKAY = 0,
        BADARG
}
```

The following will increment the enumeration value by 32.

```
enum (0x20) {
        ErrorClass0 = 0,
        ErrorClass1,
        ErrorClass2
}
```

This may be useful to define constants for bit-fields. The following example shows usage for a bit-field at bit position 7.

```
enum (0x80) {
        GFX_POINT = 0,
        GFX_LINE,    // will equal 0x080
        GFX_RECT     // will equal 0x100
}
```

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.");
}
```

gcnew (not working yet)

The new operator generates a call to the run-time library function __gcnew(). __gcnew() takes the size and type of the object and a pointer to the function's object list. __gcnew() will allocate storage for the object on the heap, then add the object to the list of objects created in the function. Objects allocated with gcnew that are not deleted before the function exits are added to the garbage collection list.

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 {
        }
}
```

new (not working yet)

The new operator generates a call to the run-time library function __new(). __new() takes the size and type of the object and a pointer to the function’s object list. __new() will allocate storage for the object on the heap, then add the object to the list of objects created in the function. Objects allocated with new that are not deleted before the function exits are added to the garbage collection list.

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:

```
;               switch(btn; naked) {
                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.

throw

Throw acts in a similar fashion to the return statement. Throw returns to the latest catch handler. The latest catch handler does not have to be defined in the current routine or a previous routine. Throwing an exception will walk backwards up the stack to the most recently defined catch handler. Unlike c++ throw does not automatically destroy objects created in the subroutine or method.

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;

try { }

Try defines a try – catch block. A block of statements followed by a series of catch statements. Try causes the compiler to output code to point to the catch block of the try statement. This pointer will be used by subsequent throw statements.

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 CC64. 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.

name mangler

The name mangler generates unique names for methods so that there is no name clash for overloaded or derived methods. A hash string representing the type, method parameter and return value types is added to the method name.

&&&

The &&& operator indicates to the compiler that a safe optimization is to generate code that executes both sides of the operator then uses an 'and' operation to determine the result. This may eliminate branches.

|||

The ||| operator indicates to the compiler that a safe optimization is to generate code that executes both sides of the operator then uses an 'and' operation to determine the result. This may eliminate branches.

??

The ?? ternary operator indicates to the compiler that a safe optimization is to generate code that executes both sides of the operator then select between the result. This may eliminate branches.

Character Constants

String constants may be pre-pended with one of ‘B’, ‘C’, ‘H’, or ‘W’ to indicate the size of encoded characters. ‘B’ = byte, ‘C’ = char (16-bit), ‘H’ = halfword (32-bit), and ‘W’ = word (64-bit).

```
Example: The following string is encoded as bytes:
B”? = display help”
```

String constants may also be placed inline with code using the letter ‘I’ as a prefix. Inlined string constants are automatically 16-bit encoded to remain aligned with instructions.

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 CC64:

```
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 CC64 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.

Exception handling

CC64 stores the current exception handler address in register r28. As the program runs and try / catch blocks are encountered register r28 is updated to match the current exception handler address. All a throw statement does then is load registers with the exception type and exception value then jump to wherever r28 points.

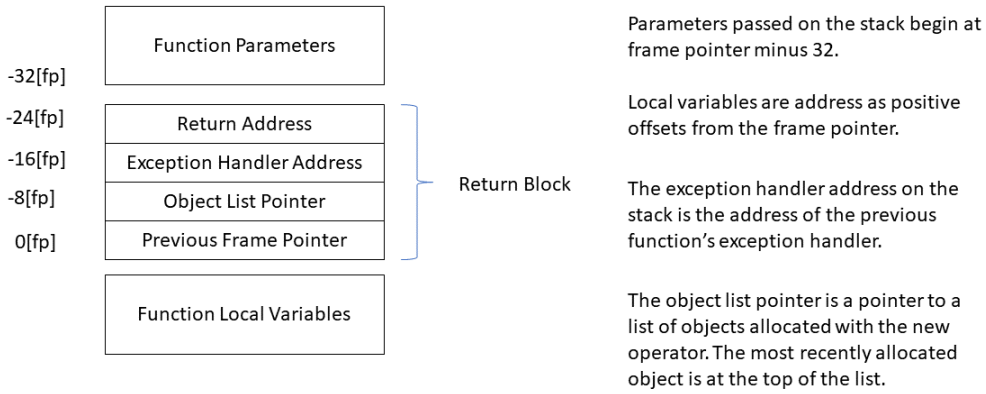
On function entry, the exception handler address of the calling function (r28) is stored on the stack, and then register r28 is loaded with the address of the default exception handler for the function. The default exception handler does nothing more than move the stacked exception handler address into the link register and then jump to the function return code. This causes an unhandled exception to unwind the stack just as a return would, then return to the caller’s exception handler address rather than the normal return address.

Because of the simplicity of the exception handling mechanism objects created in the function are not automatically destroyed. That means it's necessary to keep track of which objects got created and destroy them in the catch handler.

Garbage Collection

If a function or method uses the new operator, then a call is made to the run-time library function __AddGarbage() when the function returns. The __AddGarbage() function moves objects off the function's object list onto the garbage collector's list.

Return Block



- Deleting objects in the reverse order that they were allocated in will give the best performance, because then the top object on the object list can be deleted without having to search the object list.
- When the function exits any objects remaining on the object list are re-listed on the garbage collection list.