C64 Language Reference

Limitations:
C64 is limited to three dimensional arrays. The compiler is now coded to support up to 20 dimensions however this feature is untested.

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)
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
class variables

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[pxr]	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 -o by itself disables all optimizations done by the compiler
-p<processor>	generate code for the specified processor. -pFISA64 for the FISA64 processor -pRaptor64 for the Raptor64 processor -pThor for the 64 bit Thor processor -pThor32 for the 32 bit Thor processor If unspecified code for the Table888 processor is generated.
-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.

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.

Built In Types

byte – eight bit signed value. Generally the smallest memory operand that the instruction set can handle.

char – sixteen bit signed value.

int – sixty-four bit signed value

float – sixty-four bit double precision floating point value

All built in types are signed values by default. The “unsigned” keyword may be applied to a type to switch it to unsigned values.

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.

__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);
```

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)
{
}
```

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.

```
nocall myfunction()
{
        asm {
        }
}
```

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
        }
}
```

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
          }
}
```

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
        }
}
```

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.");
}
```

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”);
}
```

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;

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.

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>
    }
}
```

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.

Classes

Classes are defined using the “class” keyword. Classes may inherit from other classes, however only single inheritance is supported.

Class members may be designated as private which means they are only accessible to methods of the class. Trying to access private members from outside the class typically results in a number of error messages.

The “unique” keyword is used to create a class member for which there is only a single instance. Unique members are shared by all instances of the class. (The keyword static may also be used).

The compiler relies on the assembler supporting long variable names if classes are in use. The class name references aren’t hashed so using a deep class hierarchy will result in extremely long names in assembler code. The compiler simply adds the name components together separated by underscores. It’s best to keep class names short.

Differences from C++

Template classes are not supported.

Like many newer languages, multiple inheritance is not supported. This is a little used and confusing feature.

new

Storage for variables may be allocated using the new operator. new makes a call to the library function malloc. Unlike C++ new does not call an object constructor, it simply allocates memory according to the size of the type specified, or the given expression. The variable must be explicitly initialized to desired values. It is suggested that the allocation be wrapped up in a wrapper function as part of an object factory. A typical name for this function is Make(). new returns a pointer to memory allocated for the object.

new type
new (expression)

delete

delete deallocates the storage allocated by the new operator. It makes a call to the library routine "free". It uses the address supplied by the new operator. delete does not call an object destructor. The object must be explicitly destructed before being deleted.

delete expression; // Expression is a cast expression

Activating the Compiler Name Mangler

By default the compiler does not mangle names, the name output by the compiler is the name given in the program. Since names are not mangled the names of all functions in a program must be unique including method names in class declarations. To activate the compiler's name mangling facility use the "using" keyword as shown below:

using name mangler; // use compiler name mangling facility

Activating the name mangling facility allows the use of overloaded function definitions. The actual function call is determined from the class and function parameters in addition to the name.

Name mangling may be turned off to allow for pure C code via the using statement:

using real names; // stop using name mangling