C64 Language Reference

C64 supports an extended 'C' language compiler. In addition to the standard 'C' language C64 adds the following:

run-time type identification (via typenum())
exception handling (via try/throw/catch)
spinlocks (via spinlock/lockfail/spinunlock)
interrupt functions
multiple case constants eg. case '1','2','3':
inline 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

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.

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);
}
```

spinlock

A spinlock acts to guard a section of program code against re-entry. The spinlock will block a second thread from executing a protected section of code, until the first thread is finished with it. A spinlock waits for a semaphore variable to become available. There are two forms of the spinlock statement, one with a timeout and one without. If no timeout is specified then the spinlock could stall the program because it is waiting forever for a semaphore to become available.

```
char semphore1;

main()
{
        spinlock(semaphore1) {        // this will wait potentially forever
                printf(“hi there”);   // this code is protected by the spinlock
        }
}
```

lockfail

Lockfail is used with a spinlock statement when the spinlock specifies a timeout. It acts a bit like a catch statement.

```
char semphore1;

main()
{
        spinlock(semaphore1, 2000) {          // this will try 2000 times
                printf(“hi there”);   // this code is protected by the spinlock
        }
        lockfail {
                printf(“the spinlock failed.”);
        }
}
```

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 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 {
        }
}
```

interrupt

The interrupt calling convention causes the compiler to output code to save all the registers on the stack, and restore them at the end of the interrupt function. The interrupt function is also exited using an interrupt return 'iret' instruction rather than a 'ret' instruction.

```
interrupt myIRQ()
{
}
```

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;
```