C for Blitzers

by Jan@Varho.Org

This tutorial attempts to show the basics of working with C in your BlitzMax programs. BlitzMax manual/help has a section titled "Interfacing with C and other languages", but I find it short and confusing.

0. Introduction

Background

You should know the basics of C to understand what this is all about. If you know Java or some other language that has a C-like syntax, you should be able to follow. I recommend the WikiBook on C, if you need to know more about the language itself.

To compile C on Windows you need to install MinGW as detailed elsewhere. On other platforms GCC is likely already installed, so you don't have to do anything.

On BlitzMax side you should understand pointers: BlitzMax has two kinds of by-reference variable types: the Ptr-types and the Var-types. With the former you need to explicitly create the pointer using VarPtr or a cast, with Var-types the conversion is done implicitly by the compiler. Both are useful for different purposes. See the docs for syntax.

Style

I always use SuperStrict and Framework. SuperStrict is important when you reference C programs, because Strict- and non-Strict-mode assume that all functions return an Int. Framework just makes compiling a lot faster, so there's no reason not to use it.

(I use BRL.StandardIO for command line programs, MaxGUI.MaxGUI for GUI programs and BRL.Max2D for graphics. That's just to make sure I can import files across projects.)

1. Hello world with printf

Let's begin with the following C code that you wish to call from your BlitzMax program:

```
void Say(char *str)
```

```
{
	printf(str);
}
```

To use the function from within BlitzMax you need to import the file. C files are imported just like BlitzMax files, which is rather handy. However, Import only tells BMK to link the file - you still need to declare the function in BlitzMax using Extern. Here's how:

```
Import "tutorial1.c"

Extern
	Function Say(str:Byte Ptr)
End Extern

Say "Hello World!".ToCString()
```

Note the case! Unlike BlitzMax, in C case matters. Since the C-function was named Say, the declaration in Extern must match that. Later on in your BMX-code you don't have to worry about it anymore.

Ok, that works, but it's messy to convert the string to C-format and a real program would need to MemFree the string after use. Thankfully there's an undocumented feature in BlitzMax that helps:

```
Extern
	Function Say(str$z)
End Extern

Say "Hello World!"
```

That type tag $z works just as $ or :String would on the BlitzMax side, but the compiler converts the string to C-format before passing it to the function. It also automagically handles memory management, so you don't need to worry about leaks from unfreed memory.

Here's the final program:

```
' tutorial1.bmx
SuperStrict
Framework BRL.StandardIO
Import "tutorial1.c"

Extern
	Function Say(str$z)
End Extern
```

```
Say "Hello World!"
```

```
// tutorial1.c
void Say(char *str)
{
	printf(str);
}
```

2. Hello world with Print

Next, suppose you need to call a BlitzMax function from C. To keep things simple we'll create a function that works just like printf, so we can just replace the call in the above example:

```
Function PrintCString(s:Byte Ptr)
	Print String.FromCString(s)
End Function
```

Next, let's modify the C-code to call that function. For that we need to declare the function in C. Again, case is important. The declaration would normally go to a header file (.h), but here we do it in the same file:

```
void bb_PrintCString(char *s);

void Say(char *str)
{
	bb_PrintCString(str);
}
```

Note the bb_ -prefix. That is added to main program functions by the Blitz compiler to prevent name clashes. If you need to call a function from a module, you need to use something like brl_standardio_Print instead.

(Calling the "main" part of Blitz files is also possible using _bb_main for the main program or __bb_standardio_standardio for a module. Perhaps more usefully MemFree and MemAlloc can be called using bbMemFree and bbMemAlloc.)

Now the program should work just as before, but with Print instead of printf doing the job. Notice that I've decided to call the Say function SaySomething instead to show the syntax:

```
' tutorial2.bmx
```

```
SuperStrict
Framework BRL.StandardIO
Import "tutorial2.c"

Extern
	Function SaySomething(str$z)="Say"
End Extern

Function PrintCString(s:Byte Ptr)
	Print String.FromCString(s)
End Function

SaySomething "Hello World!"
```

```
// tutorial2.c
void bb_PrintCString(char *s);

void Say(char *str)
{
	bb_PrintCString(str);
}
```

3. Some math and Vars

Ok, now let's attempt to do something remotely useful for a change. Using math functions in BlitzMax can be slow compared to C, due to the lack of compiler inlining. This isn't usually a big problem, but if you need several Sqrs and trigonometric functions over a large set of data, you can do it faster with C.

To keep the code simple we'll just need to take a vector of the form (x,y,z), normalize it and return the length. (This is way too simple to actually require C, though.)

Here's the C code:

```
float Normalize(float *x, float *y, float *z)
{
	float length = sqrt( x[0]*x[0] + y[0]*y[0] + z[0]*z[0] );
	float m = 1 / length;
	x[0] *= m;
	y[0] *= m;
```

```
	z[0] *= m;
	return length;
}
```

Simple enough: the notation x[0] means the data immediately at location x - exactly as in BlitzMax. Now the straightforward way would be to declare the arguments as Float Ptr on BlitzMax side, but that would require the use of VarPtr or something similar when passing variables. Instead we'll use Float Var:

```
Extern
	Function Normalize:Float(x:Float Var, y:Float Var,..
	                                                z:Float Var)
End Extern
```

Now the variables can be passed like to any function (but note that they have to be variables, literal numbers or expressions won't do):

```
Local x:Float = 11.9, y:Float = -3.5, z:Float = 0.1
Print Normalize(x,y,z)
Print "("+x+","+y+","+z+")"
```

And here's the whole thing:

```
' tutorial3.bmx
SuperStrict
Framework BRL.StandardIO
Import "tutorial3.c"

Extern
	Function Normalize:Float(x:Float Var, y:Float Var,..
	                                                z:Float Var)
End Extern

Local x:Float = 11.9, y:Float = -3.5, z:Float = 0.1
Print Normalize(x,y,z)
Print "("+x+","+y+","+z+")"
```

```
// tutorial3.c
float Normalize(float *x, float *y, float *z)
{
	float length = sqrt( x[0]*x[0] + y[0]*y[0] + z[0]*z[0] );
	float m = 1 / length;
	x[0] *= m;
```

```
	y[0] *= m;
	z[0] *= m;
	return length;
}
```

4. Working with objects

There are several different use cases with objects and Blitz/C interface. In the simplest case you only need to pass an object to the C side, maybe to return it again later, but without the need to access the data.

Passing an object from BlitzMax works just like passing a primitive. To pass and accept Blitz-strings (strings are objects) you could declare external functions like this:

```
Extern
	Function StringTaker(str:String)
	Function StringGiver:String()
End Extern
```

Nothing special there yet. To accept it on the other side, your C functions must be declared as using a byte (char) pointer:

```
void StringTaker(char *str);
char * StringGiver();
```

However, this approach does not let you do anything much with the object. Sure, you can store it somewhere and return it again from elsewhere, but that's about it. (Note: always make sure the object is not garbage collected by also retaining a reference on Blitz side.)

Instead, you can pass a pointer to the object's data by casting the object to a byte pointer in Blitz. Let's create a simple vector type to test this:

```
Type TVector
	Field x:Float
	Field y:Float
	Field z:Float
End Type

Extern
	Function NormalizeVector:Float(v:Byte Ptr)
End Extern
```

Ok, now the object is implicitly converted by the Blitz compiler to a pointer to the first field when passed. And we don't even have to cast explicitly, since Object->Byte Ptr casts are automatic. So we can just use it like this:

```
Local v:TVector = New TVector
v.x = 4.2
v.y = 2.4
v.z = 42.0

Print NormalizeVector(v)
Print "("+v.x+","+v.y+","+v.z+")"
```

Now, how about the implementation? We can almost copy the Normalize function used earlier. We just need to remember that y and z fields are offset from the x field. Like so:

```
float NormalizeVector(float *v)
{
	float length = sqrt( v[0]*v[0] + v[1]*v[1] + v[2]*v[2] );
	float m = 1 / length;
	v[0] *= m;
	v[1] *= m;
	v[2] *= m;
	return length;
}
```

I used a float pointer instead of a byte pointer, since both are just memory addresses. This is a good example of how little the compilers hold your hand when doing cross-language stuff: no warnings are emitted even though function declarations differ in type.

That's it. The whole program follows. As always with pointers, you must be careful not to access out of bounds. Also, as I mentioned above, you must make sure the object is not garbage collected by retaining a Blitz reference:

```
' tutorial4.bmx
SuperStrict
Framework BRL.StandardIO
Import "tutorial4.c"

Type TVector
	Field x:Float
	Field y:Float
	Field z:Float
End Type

Extern
```

```
      Function NormalizeVector:Float(v:Byte Ptr)
End Extern

Local v:TVector = New TVector
v.x = 4.2
v.y = 2.4
v.z = 42.0

Print NormalizeVector(v)
Print "("+v.x+","+v.y+","+v.z+")"
```

```
// tutorial4.c
float NormalizeVector(float *v)
{
      float length = sqrt( v[0]*v[0] + v[1]*v[1] + v[2]*v[2] );
      float m = 1 / length;
      v[0] *= m;
      v[1] *= m;
      v[2] *= m;
      return length;
}
```