BS BLAST SCRIPT
A BURST COMPATIBLE SCRIPT INTERPRETOR

BLAST Language Reference

BLAST Overview - Namespace (doxygen) - Examples

BlastScript

-A blast script is a collection of statements seperated by dotcomma's and grouped by parenthesis.

Blastscript sample:

```
#input position float3 0 0 0
position.x = min(100, position.x + 0.001);
```

Full C# GameObject example:

```
public class Single2 : MonoBehaviour
{
    BlastScript script;

    void Start()
    {
        Blast.Initialize();

        // prepare the script once
        script = BlastScript.FromText("#input position float3 0 0 0\n position.x = min(100, position.x + 0.001);");

        BlastError result = script.Prepare();
        if (result != BlastError.success)
        {
            Debug.LogError($"Error during script compilation: {result}");
            script = null;
        }
    }

    void Update()
    {
        if (script == null) return;

        // the each frame, update current position
        script["position"] = (float3)transform.position;

        // execute the script
        BlastError result = script.Execute();
        if (result == BlastError.success)
        {
            // and set value
            transform.position = (float3)script["position"];
        }
        else
        {
            Debug.LogError($"Error during script execution: {result}");
        }

    }
}
```

Statement Keywords

keyword	description
case	switch case compound
default	default switch case compound
else	else compound of if then else statement sequence
if	if statement
input	defines an input to the script
for	for loop
output	defines an output of the script
return	return; terminate execution
switch	switch statement
then	then compound of if then else statement sequence
while	while loop
yield	yield execution
function	inline functions

Literal Keywords

Keyword	Value
false	0.0
true	default: 1.0, anything other then 0.0 matches to true

Operators

Seperators: ; , ( )

Arithmetic: `+ - * /`

Boolean: `& | ^ !`

Assingment: `=`

Comparisons: `= < <= > >= !=`

Indexing: .[x|y|z|w|r|g|b|a]

Constant data

Blast uses bytecode operations to index a datasegment for variable and stack data. Any constant value that is not internally mapped to a bytecode will be packaged in the datasegment. Blast map's several often used values like 0, 1, 2, pi and others to byte sized opcodes, this can save a lot of room in the datasegment depending on how well the scripts match to the constant data. For maximum efficiency users should overwrite blasts default constant table with their own.

If for example blast is used to control a statemachine its state-ids could be stored in constants potentially saving many bytes in the compiled package and datasegment.

Defaultly mapped constants:

```
                 case blast_operation.pi: return math.PI;
                case blast_operation.inv_pi: return 1 / math.PI;
                case blast_operation.epsilon: return math.EPSILON;
                case blast_operation.infinity: return math.INFINITY;
                case blast_operation.negative_infinity: return -math.INFINITY;
                case blast_operation.nan: return math.NAN;
                case blast_operation.min_value: return math.FLT_MIN_NORMAL;
                case blast_operation.value_0: return 0f;
                case blast_operation.value_1: return 1f;
                case blast_operation.value_2: return 2f;
                case blast_operation.value_3: return 3f;
                case blast_operation.value_4: return 4f;
                case blast_operation.value_8: return 8f;
                case blast_operation.value_10: return 10f;
                case blast_operation.value_16: return 16f;
                case blast_operation.value_24: return 24f;
                case blast_operation.value_32: return 32f;
                case blast_operation.value_64: return 64f;
                case blast_operation.value_100: return 100f;
                case blast_operation.value_128: return 128f;
                case blast_operation.value_256: return 256f;
                case blast_operation.value_512: return 512f;
                case blast_operation.value_1000: return 1000f;
                case blast_operation.value_1024: return 1024f;
                case blast_operation.value_30: return 30f;
                case blast_operation.value_45: return 45f;
                case blast_operation.value_90: return 90f;
                case blast_operation.value_180: return 180f;
                case blast_operation.value_270: return 270f;
                case blast_operation.value_360: return 360f;
                case blast_operation.inv_value_2: return 1f / 2f;
                case blast_operation.inv_value_3: return 1f / 3f;
                case blast_operation.inv_value_4: return 1f / 4f;
                case blast_operation.inv_value_8: return 1f / 8f;
                case blast_operation.inv_value_10: return 1f / 10f;
                case blast_operation.inv_value_16: return 1f / 16f;
                case blast_operation.inv_value_24: return 1f / 24f;
                case blast_operation.inv_value_32: return 1f / 32f;
                case blast_operation.inv_value_64: return 1f / 64f;
                case blast_operation.inv_value_100: return 1f / 100f;
                case blast_operation.inv_value_128: return 1f / 128f;
                case blast_operation.inv_value_256: return 1f / 256f;
                case blast_operation.inv_value_512: return 1f / 512f;
                case blast_operation.inv_value_1000: return 1f / 1000f;
                case blast_operation.inv_value_1024: return 1f / 1024f;
                case blast_operation.inv_value_30: return 1f / 30f;
                case blast_operation.inv_value_45: return 1f / 45f;
                case blast_operation.inv_value_90: return 1f / 90f;
                case blast_operation.inv_value_180: return 1f / 180f;
                case blast_operation.inv_value_270: return 1f / 270f;
                case blast_operation.inv_value_360: return 1f / 360f;
```

Assignments | Expressions

An expression is a set of literals and identifiers combined with operators. Parenthesis can be used to influence the order of operation.

```
// numeric expressions
f = 1 * 10 * 3 * (3 + 4 * 2);
f = 10 + 5 / 1 * 2;
f = 1 * 4 + 3;
f = 4 * -9.3

// boolean expressions:
b = 1 & 0 | 1;
b = -1 & 0 | 123.22;
b = true & false | true;
```

Vectors are defined as sets of values enclosed in parenthesis without operators combining them. A comma is used to seperate elements to remove any ambiguity when using negative constants.

Inline vector definition: `(1.2 2.1, -3.3)`

```
a = (1 2 3 4) * (5 6 7, -8);
```

To get the component of a vector it can be indexed: `a.x`

Built in functions

- note: the optimizer will replace sequences with its equivevalent function whenever possible providing shorter code and faster execution due to reduced control flow.

function	description	parameters	returns	examples
abs	get absolute value of operand	1 numeric vector	vector	
adda	add all operands in sequence	n numerics of equal vectorsize	vector	
all	returns true if all arguments are true	n value of n vectorsizes	scalar	
any	returns true if any argument is true	n values of n vectorsizes	scalar	
atan	unity.mathematics.atan	any vectorsize	vector	
atan2	unity.mathematics.atan2	any vectorsize	vector	
ceil				
clamp				
ceillog2				
ceilpow2				
cos	unity.mathematics.cos	any vectorsize	vector	
cosh	unity.mathematics.cos	any vectorsize	vector	
cross	unity.mathematics.cross	vector 3	vector 3	
csum	unity.mathematics.csum	any vectorsize	scalar	
call	explicetly call a function		call	
debug				
debugstack				
degrees				
diva	divide all operands by eachother in sequence	any vectorsize	vector	
dot				

function	description	parameters	returns	examples
exp				
exp10				
floor				
floorlog2				
fma				
fmod				
frac				
lerp				
log10				
log2				
logn				
max				
maxa	get max value from operands	any vectorsize	vector	
min				
mina				
mula	multiply all operands in sequence	any vectorsize	vector	
nlerp				
normalize				
pow				
radians				
random	generate a random number			
remap				
rsqrt				
saturate				
seed	seed the random number generator			
select				
sin				
sinh				
slerp				
sqrt				
suba	substract all operands from eachother in sequence	any vectorsize	vector	
tan				

function	description	parameters	returns	examples
trunc				
unlerp				

Inline Functions

Blast allows the user to define inline functions, a function consists of the function keyword, a parameterlist (that even if empty may not be ommited) and a body:

```
function f1(a, b)
(
   c = a + b;
   return (a + b) / c;
);
```

Blast functions will however work differently then in for example C#. There is NO scope as the function is inlined at the callsite. It operates directly on the used variables and it does not make any data copies, making all parameters passed by reference. Returning data will also work directly on the same data elements. Assigning `c` within a function assigns `c` in global scope.

Inlined Functions may call other inline functions.

Inlined Functions may not declare new variables

This should be viewed as a utility function, for recurring needs please consider implementing user defined external function calls that can be called natively.

External Function Calls

Blast uses function pointers to connect to other parts of its environment, these will be supplied with the following data by the BLAST engine:

data	source	description
engine	blast	pointer to engine data, contains constants, functionpointers and randomizer root.
environment	user	optional pointer to native data containing data to be used by all scripts
caller	user	optional pointer to native data containing all data supplied by the callsite
parameters	blast	blast will call the external function using the parameters supplied to it by script

supports only single floats, future versions will expand on this

Blast will use all components of the supplied input parameters to satify the functions parameter list:

```
 a = (1 1 1);
external(a);
```

Is equivalent to:

```
 a = (1 1 1);
external(a.x, a.y, a.z);
```

BlastFunction Attribute

An attribute can be used to attach to static methods, blast will reflect the loaded assemblies and attempts to register all static methods with this attribute on startup.

```
        [BurstCompatible] [BlastFunction]
        static public float AttributeTest1(float b)
        {
            return b * 2;
        }

        [BurstCompatible] [BlastFunction(name: "test")]
        static public float AttributeTest2(float b)
        {
            return b * 2;
        }
```

[BS1] Supported prototypes

```
    // full paramater list (including enviroment data), up until 16 float params
    public delegate float BlastDelegate_f1(IntPtr engine, IntPtr data, IntPtr caller, float a);
    public delegate float BlastDelegate_f11(IntPtr engine, IntPtr data, IntPtr caller, float a, float b);
    public delegate float BlastDelegate_f111(IntPtr engine, IntPtr data, IntPtr caller, float a, float b, float c);
    // ......
    public delegate float BlastDelegate_f1111(IntPtr engine, IntPtr data, IntPtr caller, float a, float b, float c, float d...... ,

    // short external defines without environment pointers
    public delegate float BlastDelegate_f0_s();
    public delegate float BlastDelegate_f1_s(float a);
    public delegate float BlastDelegate_f11_s(float a, float b);
```

Data Synchronization

Blast uses the input and output keywords to define input or output variables. These will be prepared in the compiled package for fast IO syncs, the sync method however depends heavily on the packaging mode and its usage. Samples for each mode (Normal, SSMD, Entity) will be provided shortly;

Basic Use

Use methods provided by the BlastScript class to read or write to script variables exposed via input and/or output, it is not necessary to use input nor output defines but doing so forces their memoryorder regardless the code written and should be considered good practice, the can be omitted but you will have to directly write to the datasegments to set variables by name.

BlastScript:

```
#input a float3 3 3 3
#input b float 3

a = b * a;
```

Unity:

```


// create script, set a
BlastScript script = BlastScript.FromResource("/bs");
script.Set("a", new Vector3(4, 4, 4));
script.Set("b", 8);
script.Execute();

Vector3 result = (Vector3)script["a"];
```

Entity/IComponentData

The script's datasegment is directly mapped against an IComponentData record:

For a script defined as:

```
#input a float3 3 3 3
#input b float 3

a = b * a;
```

The IComponentData should be like:

```
public struct data : IComponentData
{
  public float3 ValueA;
  public float ValueB;
}
```

The script can then very efficiently use this as a datasegment.

Yield and Entities

If yield is used, all memory used for script execution must map to the IComponentData struct. This includes any stackmemory that is used: yield reserves 20 bytes in each segment to allow it to store its internal execution state.

SSMD

The scripts datasegment is packed as with entities but not matched to any structure, it might also not be complete depending on any constant data in it. In SSMD mode there is no stack allocated in the datasegment and in the future it will inline any constant data in the code datastream to maximize memory efficiency when running many billions of scripts each second.

Yield

Yield is not supported.

Data Validation

During development we have the need to test a lot and there is some support for automatic testing in the form of validation script defines: `#validate a 1` These allow the script to set values that blast can match to the output of the script given default input.

Blast can (depending on compilersettings) then validate the script during compilation in the same run it uses to determine the stackspace it needs to reserve in the compiled package. It proved extremely usefull during development and it possibly can catch bugs early in deployment.

Example script with validation defines:

```
#define result_1 11111
#define result_2 22222

a = 10;
b = 2;

switch(a > b)
(
  case 1:
  (
    e = result_1;
  )
  default:
  (
    e = result_2;
  )
);

#validate e 11111
```