Scripting (Torque Script)

Concepts and Terminology

To Script or Not to Script? (A rant of sorts…)

I am guessing that you, the reader, are an experienced game player, and/or perhaps have had some experience in game or mod writing. In any case, you are likely familiar with the fact that most modern games offer a method of 'programming' game behavior via some sort of scripting language. I am willing to state, that having this feature is now a requirement for any game[1] wishing to be successful. Given that, you might ask, "What can it be used for and how will that help my game be a success[2]?" Lets list a few things that game scripting can be used for in games:

- **Writing non time-critical functionality** – Scripts can be used to implement functionality which is not going to be executed frequently and/or does need to be blazingly fast. In other words, I wouldn't suggest writing a render-pipeline in script, but writing code to support GUI responses and simple NPC movement would be OK.
- **Patching** – Great, so your game is released and you just shipped your ten-thousandth copy, when one of your testers approaches you and say, "*Hey Bob. Did you get that bug report I sent in? You know, I only ask because, uh… it still happens in the Gold copy*" Now what do you do? Perhaps, you provide a scripted patch? If there is tight integration between your engine and scripting language, you very well might be able to over-ride certain engine features via script. Whew!
- **Debugging** – Another obvious use for scripting is as a debugging tool. The key reason this is valuable is because you can get rapid feedback with a low (implementation) cost.
- **Test-benches** – Along the way, as you are writing your game, you may find yourself wanting to test some new 'snazzy' feature you put into the core engine code. Why not write a test-bench to exercise the feature using scripts?
- **Prototyping** – Guaranteed, somewhere along the way, you will find yourself wanting to try out some cool new idea. Unfortunately, to try it you have to modify the engine and somehow integrate it with the rest of the code. Or, do you? If it is amenable to scripting, why not script it first, tweak it there, and then if you need to, move it into the engine?
- **Partial-Mods (Game Customization)** – This is a very familiar concept which you see in action a lot. An excellent example of this is the plethora of partial-mods for Tribes™ 2. If you've played T2, then you have at least seen, if not

[1] Besides perhaps single player games and console-only games.
[2] Now scripting alone won't make your game successful, but having it will help.

played, such partial-mods as: War 2002, Renegades, and Team Aerial Combat (which started as a mod for Starseige Tribes™).

- **Total-Mods** – If you've every played Half-Life™, then you know what I'm talking about here. HL is without doubt the most MODed game of all time. In addition to the thousands of partial-mods out there, you will find just as many (if not more) new games, based on the HL engine, but written to a comptely different genre using new models and new scripts.
- **Game Creation** – So, given total-mods, it isn't much of a stretch to imagine that one could create an entire game via script. I'd say that this not impossible, but a bit of a stretch. For performance sake alone, I do not suggest this. However, if performance is not your number one limiting factor, more power to you.

Features We Need

Given that we accept scripting is useful to us, what should we be looking for in a scripting language? i.e. What functionality should it provide to us? As a bare minimum, a scripting language, for use in a game, should provide the following features:

- **Familiar and Consistent Syntax** – Ideally, the syntax of the scripting language is familiar, meaning it is similar to the syntax of a language we, as programmers, are already familiar with, for example C or C++. Also ideal is that it be consistent, meaning the same rules which apply to the familiar programming language also apply to the scripting language.
- **Provides complete set of basic operations and constructs** – The scripting language should provide a tool-kit of basic operations (addition, subtraction, etc). Additionally, it should provide the standard constructs (if-then-else, for, while, etc).
- **Provide access to Engine-Structures** – This is a critical feature. If the scripting language is going to be of any use to us at all, it must provide some kind of interface giving us a means of interacting with the engine and engine-structures. In other words, we should have access to the render engine, we should be able to create and delete objects, and we need complete control of and access to the I/O sub-system.

Some other (very) nice to have features are:

- **Provides Object Oriented Functionality** – If we're going to ask for features, why not ask that the scripting language provide us some of the features found in object oriented languages:
  - Encapsulation – Provide a means of limiting access to code and data.
  - Inheritance – Provide a means of creating new 'objects' from engine objects and/or scripted objects.
  - Polymorphism – Allow us to over-ride the default behavior of derived object code, whether the object be derived from engine objects or other scripted objects.
- **Provide 'On-demand' Loading and Scoping** – Why have all our code in memory at once when we can load it as needed? Also, why not enable us to load and unload functionality as needed, simultaneously over-riding prior functionality and replacing it as necessary.
- **Provide a means of 'speeding-up' our scripted code** – Scripted code is, by default interpreted. Interpreting is slow… A feature than many common scripting languages (PERL, TCL, VB Script, Java, …) provide is the ability to 'compile' scripts into a pcode. This pcode is then 'executed' on a virtual machine. The benefits of this are size and speed. Pcode is (normally) smaller than and executes faster than interpreted code.

Use a Standard Scripting Language or 'Roll Your Own'

If you've ever attempted to write your own game-engine and decided to implement some kind of scripting interface, you've been challenged with this question, "Do I integrate a well known scripting language like PERL or Java, or do I go the long way and write my own scripting language?" Both of these approaches have their own benefits and drawbacks. The short list for each would be:

Benefit/Drawback	**Standard Language (Ex: PERL)**	**Roll Your Own**
Familiarity	Users are probably already familiar with this.	No dice. You'll have to model it after a common language for it to be familiar.
Documentation	Yep, lots of it.	No dice again. Gotta write that too.
Engine integration	Ouch! Languages like PERL, although definitely amenable to integration, were written to be generic (read as, "a pain to integrate").	Hooya! Finally. This is the number one reason to roll your own. Of course, that doesn't make it easy, but it does give you total control.

What about Torque Script?

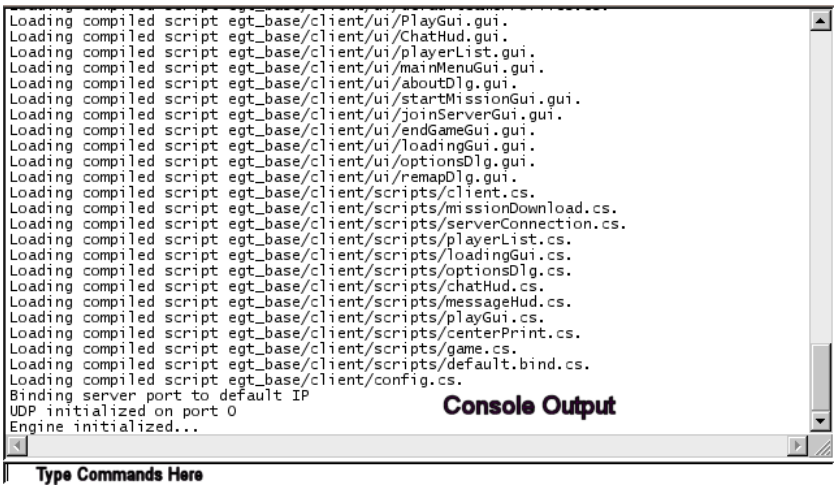
At this point, you may be falling asleep, so I'll try not to go on and on too much longer. So far, I have been trying to build up to a discussion of the scripting language provided with Torque. *Torque Script*, as the Torque scripting language will be refered to from here on, is a 'Roll Your Own' style scripting language with a syntax very similar to C++. It provides all of the features listed above, including those on the 'would be nice' list. The following sections will describe all of these features and provide examples to clarify concepts. Additionally, there are scripting references in the appendix.

The Console (GUI) and Simple Scripts

Over the next few pages we will investigate specific Torque Script features. To do this we will either:

- Use the console and test examples directly on the command line, or
- Load a test script via the console, and execute it there.

So, how do we start the console GUI? Easy. First, start the SDK, now hit the tilde key '~'. You should now see the console:



An important command we'll be using a lot is the echo() command. It has the following basic syntax:

```
echo(string0 [, string1 [,...,[string n]]]);
```

Any statements in a box like the following can be cut and then pasted (CTRL + V) into the console. Try these:

```
echo("Torque Rocks");
echo (1+1);
echo(16385 | 32768);
```

Convert the result of the last one to HEX (use a calculator if you must), and what do you get? C001 !!!

Torque Script Features

- **Type-less** – In Torque Script, there is a concept of integers, floats, or strings but, variables can be used interchangeably and will be converted as necessary. Torque script provides several basic literal types which will be described below.

```
if("1.2" == 1.2){ echo("same"); } else{
echo("different");};
```

- **Case-insenstive** – Be careful, Torque Script ignores case when interpreting variable and function names.

```
$a = "An example";
echo($a);
echo($A);
```

- **Statement termination** – Statements are terminated with a semi-colon (**;**).

```
$a = "This is a statement";
```

- **Full complement of operators** – The complete list is in the appendix, but Torque Scripts provides all the basic operators and a few advanced, including arithmetic, relational, logical, bitwise, assignement, string, and access.

- **Full complement of Constructs** – As with any robust language, Torque Script provides the standard programming constructs: if-then-else, for, while, and switch.

```
for($a=0;$a<5;$a++) {echo($a); };
```

- **Functions** – Torque scrcipt provides the ability to create functions with the optional ability to return values. Parameters are passed by-value and by-reference. (see 'functions' below for detailed description and examples.)

- **Provides Inheritance and Polymorphism** – Torque Script allows you to inherit from engine objects and to subsequently extend or override (morph) object methods. (see below for detailed description and examples.)

- **Provides Namespaces** – Like C++, Torque Script supports the concept of namespaces. Namespaces are used to localize names and identifiers to avoid collisions. Huh? This means, for example, that you can have two different functions named doit() that exist in two separate namespaces, but which are used in the same code. (see 'namespaces' below for detailed description and examples.)

- **Provides on-demand loading and unloading of functions** – Torque Script supports a very cool feature which allows you to load and unload functions as needed. (see 'packages' below for detailed description and examples.)

- **Compiles and executes PCODE** – As a bit of icing on the cake, the Torque scripting engine compiles scripts prior to executing them, giving a speed increase as well as providing a point at which errors in scripts can be reasonably found and diagnosed.

Variables

Variables come in two flavors in Torque Script: Local and Global. Local variables are transient, meaning they are destroyed automatically when they go out of scope. Global variables, as you would expect, are permanent. The syntax is as follows:

```
%local_var = value;
$global_var = value2;
```

Variables do not need to be created before you use them. Just use them. If you attempt to evaluate the value of a variable that was not previously created, it will be created for you automatically.

```
for(0;%a<5;%a++) { echo(%a); };
echo(%a);
```

Note: The above code creates an error message on the first iteration, because the value isn't created till the first time the loop completes. Also, %a is destroyed after the loop and therefore we only print the value '4' once.

Variable names may contain any alpha-numeric (a..z, A..Z, 0..9), as well as the underscore (_) but must start with an alpha- or an underscore. You may end variable names with a numeric, but if you do, you must be especially careful with array names. For further explanation, see of arrays below.

Lastly, local and global variables can have the same name, but contain different values.

```
$a="EGT:: ";
for(0;%a<5;%a++) { echo($a SPC %a); };
```

Data Types

Various types of data are supported by Torque script:

- Numeric:

```
123     (decimal)
1.234   (floating point)
1234e-3 (scientific notation)
0xc001  (hexadecimal)
```

Nothing mysterious here. Torque Script handles your standard numeric types. I do believe, floating point values are stored as 32-bit values (i.e. single-, not double-precision).

- String:

```
"abcd"  (string)
'abcd'  (tagged string)
abc1234 (auto-string)
```

Standard strings, of the form "value" behave as you would expect. Try these examples:

```
echo("Hello!");
echo("1.5" + "0.5");
```

Tagged strings are special in that they contain string data, but also have a special numeric tag associated with them. Tagged strings are used for sending string data across a network. The value of a tagged string is only sent once, regardless of how many times you actually do the sending. On subsequent sends, only the tag value is sent. When printing tagged values, you have untag them. Try these examples:

```
$a="This is a regular string";
$b='This is a tagged string';
echo(" Regular string: " SPC $a);
echo("  Tagged string: " SPC $b);
echo("Detagged string: " SPC detag('$b'));
```

Note: You may find it odd that the last line shows a blank. This is because, although we have created the tagged string, it has been transmitted to us. You can <u>ONLY</u> detag a

tagged string which has been passed to you. (Tagged strings will be discussed more in the networking chapter).

So, what about these 'auto' strings. Well, I'd like to say I know why this is true, but I don't. All I know is that, contiguous statements containing alpha [and numeric] values and starting with an alpha seem to get automatically coverted to a string.

```
$a=Is_this_interesting_or_what;
echo($a);
```

- String Constant:

```
TAB     (tab)
SPC     (space)
NL      (newline)
```

There is an operator for string catenation '@' in Torque Script. It has the basic function of taking two strings and stitching them together. So some smart cookie, probably at the behest of the script writers, added a few special 'string constants' that behave in the same way. The basic syntax for these string constants is:

"string 1" *op* "string 2"

```
echo("Hi" TAB "there.");
echo("Hi" SPC "there.");
echo("Hi" NL "there.");
```

- Escape Sequence:

```
\n      (newline)
\r      (carriage return)
\t      (tab)
\c0…\c9 (colorize subsequent text)
\cr     (reset to default color)
\cp     (push current color on color stack)
\co     (pop color from color stack)
\xhh    (two digit hex value ASCII code)
\\      (backslash)
```

One of the cool and useful things that you don't find to this extent in all scripting languages is 'escape sequences'. As in C, you can create new-line and tabs using the

tried and true backslash character. Additionally, for data that is printed to the console and GUIs, you can colorize by using '*\cn*', where n is a value between 0 and 9. Additionally, you can set the colors to be whatever you would like them to be. However, there is more than one way to do this, so I'll summarize this in appendix EFM – Console Color.

```
echo("\c2ERROR!!!\c0  => oops!");
```

- Boolean:

```
true      (1)
false     (0)
```

- Array:

```
$ary_Foo[n]     (Single-dimension)
$ary_Bar[n,m]   (Multi-dimension)
$ary_Bar[n_m]   (Multi-dimension)
```

Note: A common misconception is that multi-dimensions are not supported in Torque Script. This is not true. They are supported, but use a non-standard format: comma-separation of indicies. Also, be aware that the commas are converted to underbars and the indicies are concatenated.

CAUTION 1: $a and $a[0] are separate and distinct variables. I strongly suggest using some type of naming convention so it is clear when you are dealing with an array.

```
$a = 5;
$a[0] = 6;
echo("$a == ", $a);
echo("$a[0] == ", $a[0]);
```

CAUTION 2: $ary_myarray0 and $ary_myarray [0] are the same array. Yep, it may be surprising, but Torque must figure you were lazy or something. I strongly suggest using a naming convention to differentiate variable contents. To this day, although I know why it is done, I still think it is bad..bad..bad practice to teach students to use i, j, k, etc. as array indicies. See the example below for what could happen if you use variations of these too-short and poorly named variables in your code.

```
$i0=0;
$i1[$i0]=$i0++;
$i1[$i0]=$i0++;
$i1[$i0]=$i0++;
```

```
echo("$i0 == ",$i0);
echo("$i[0] == ",$i[0]);
echo("$i1 == ",$i1);
echo("$i1[0] == ",$i1[0]);
echo("$i1[1] == ",$i1[1]);
echo("$i1[2] == ",$i1[2]);
```

- Vector:

```
"1 m n o"   (4 element vector)
```

'Vectors' are a cool pseudo data-type which you have probably been using without even knowing it. Many fields in the inspector take numeric values in sets of 3 or 4. These are stored as strings and interpretted as 'vectors'. There is a whole set of console operations (read as 'available in scripts') for manipulating vectors. Also, as mentioned, vectors are give as input to all kinds of game methods.

```
$vec_ray0  = "1.0 0.0 1.0";
$vec_ray1  = "1.0 6.0";
echo(VectorAdd($vec_ray0, $vec_ray1));
```

Note: Most vector functions only operate on vectors of three or fewer elements. I've added an appendix listing each function and showing pertinent usage details.

Operators

Because I think it would be both a waste of your time and mine to list data about operators in both the body of this guide and in the appendix, I'm going to refer you to appendix 'EFM' for operators. The only thing I'll discuss here is a few special (i.e. non-standard) operations you might otherwise stumble over.

- The ++ and -- operators are only post-fix operators (i.e. ++%a; does not work)
- Separate comparison operations for numeric and string:
  - **Numeric comparisons are of the form:**

```
==      (numeric equal)
!=      (numeric  not-equal)
etc.    (See Appendix for remainder)
```

  - **String comparisons are of the form:**

```
$=      (string equal to operator)
```

```
!$=    (string not equal to operator)
```

- **String catenation.** You can concatenate (stitch together) strings using the catenation operator.

```
@      (string catenation operator)
```

Ex: “Hello “ @ “world” becomes “Hello world”

Constructs

By constructs, I mean branching and looping structures. Torque Script supports the following:

Branching Structures

- **if-then-else**

The general structure of the if-then-else construct is:

```
if(expression) {
     statements;
} else {
    alternate statementss;
};
```

Things to know:

- brackets ‘{}’ are optional for single line statements. (Suggestion: always use them.)
- No, there is no ‘then’ statement. It is implicit.
- Compound if-then-else-if-then-… statements are perfectly legal (Suggestion: Consider a switch statement for clarity sake.).
- Don’t forget the semi-colon after the final bracket.

- **switch**

The general structure of the switch construct is:

```
switch(expression) {
case value0:
     statements;
     break;
case value1:
     statements;
     break;
…
case valueN:
     statements;
     break;
default:
     statements;
};
```

Things to know:

- switch only (correctly) evaluates numerics. There is a special construct 'switch$' for strings.
- break statements are superfluous. Torque Script will only execute matching cases.
- Don't forget the semicolon after the closing bracket.
- **switch statements <u>are not</u> faster than if-then-else. (See EFM)**

- **switch$** - This construct behaves exactly like the 'switch' construct with one important exeception. It is <u>only for strings</u>.

Looping Structures

- **for**

The general structure of the for loop is:

```
echo(string0 [, string1 [,...,[string n]]]);
```

for(expression 0; expression1; expression2) {
  statement(s);
};

Things to know:

- expression0 is usually of the form: **var** *assign-op* **value** (ex: $count = 0)

- expression1 is usually a statement of the form: **var** *compare-op* **value** (ex: $count == 5)
  - if this statement evaluates to 0 or empty-string (EFM check this), the for loop will terminate.
- expression2 is usually of the form: **var** *op* (ex: $a++)
- if you wish to exclude expression0 or expression2, you may do so, but must place something as a placeholder. I use a 0.

```
$count=0;
for(0;$count < 5; 0) {
    echo($count++);
}
```

- You may use local variables for each expression. These expressions will be automatically destroyed once the loop terminates.

```
for(%count=0;%count < 5; %count++) {
    echo(%count);
}
echo(%count);
```

- Don't forget the semicolon after the closing bracket.

- **while**

The general structure of the for loop is:

```
while(expression) {
     statements;
};
```

Things to know:

- The while loop will continue to execute until the expression evalutes to 0 or an empty-string.
- brackets '{}' are optional for single line statements. (Suggestion: always use them.)
- Don't forget the semicolon after the closing bracket.

Console Functions

By definition, as a procedural language, Torque Script supports functions. Basic console functions are defined as follows:

```
function func_name([arg0],...,[argn]) {
    statements;
    [return val;]
}
```

Console functions can take no arguments or any number of arguments separated by commas. Also, a function may return an optional value.

Things to know:

- Defining subsequent console functions with the same name as prior console functions, over-rides the previous definition permanently unless the re-definition is within a package (see packages below).
- If you call a console function and pass fewer parameters than the console function was defined with, unspecified parameters will be given an empty string as their default value.

Objects

Having covered the basics of Torque Script let us examine some more interesting details. In Torque, every item in the 'Game World' is an object. Furthermore, all script objects are created from C++ objects or datablocks. Examples would be: Player, WheeledVehicle, TSStatic, etc.

Object Creation Syntax:

```
// In TorqueScript
%var = new ObjectType(Name : CopySource, arg0, …, argn) {

<datablock = DatablockIdentifier;>

[existing_field0 = InitialValue0;]
...
[existing_field N = InitialValueN;]

[dynamic_field 0 = InitialValue0;]
...
[dynamic_field N = InitialValueN;]
};
```

This syntax is simpler than it looks. Lets break it down:

- **%var** – Is the variable where the object's handle will be stored.

- ***new*** – Is a key word telling the engine to create an instance of the following ObjectType.
- **ObjectType** – Is any class declared in the engine (must be derived from SimObject or a subclass of SimObject).
- **Name (optional)** – Is any expression evaluating to a string which will be used as the objects name.
- **: CopySource (optional)** – EFM TBD – Work out example for this. 'GuiTextProfile' is an example of this.
- **arg0, …, argn (optional)** – Is a comma separated list of arguments to the class constructor (if it takes any).
- **datablock** – Many objects (those derived from GameBase or children of GameBase) require datablocks to initialize specific attributes of the new object. We'll discuss datablocks below.
- **existing_memberN** – In addition to initializing values with a datablock, you may also initialize existing class members (fields) here.
    - Note: Any member you wish to modify must have been exposed. We'll talk abou this later.
- **dynamic_memberN** – Lastly, you may create new fields (which will exist only in Script) for your new object. These will show up as dynamic fields in the World Editor Inspector.

Let's create one object that doesn't use a datablock and one that does:

```
// create a SimObject w/o modifying any fields
$example_object = new SimObject();

// create a SimObject w/ dynamic fields
$example_object = new SimObject() {
a_new_field = "Hello world!";
};

// create a StaticShape using a datablock
// to run: exec("egt_base/…/datablock0.cs");
datablock StaticShapeData(MyFirstDataBlock) {
    shapeFile = "~/data/shapes/player/player.dts";
    junkvar = "helloworld";
};

new StaticShape() {
    dataBlock = "MyFirstDataBlock";
    position = "0.0 0.0 0.0";
    rotation = "1 0 0 0";
    scale = "1 1 1";
};
```

Handles and Names

Every object in the game is identified and tracked by two parameters:

- **Handles** – Every object is assigned a unique numeric ID upon creation. This is generally referred to as the object's handle.
- **Names** – Additionally, all objects may have a name.

In most cases, Handles and names may be used interchangeably to refer to the same object, but a word of caution is in order. Handles are always unique, whereas multiple objects may have the same name. If you have multiple objects with the same name, referencing that name will find one and only one of the objects. Generally, it is best to use handles to refer to objects. This is faster anyway (name references require a lookup).

Fields and Commands

Similar to Members and Methods in C++, are Fields and Commands in Torque Script. Objects instatiated via script may have data members (referred to as Fields) and functional Methods (referred to as Commands). In order to 'access' an object's Fields or Commands, one uses dot notation.

```
// Directly access via handle
123.field_name = value;
123.command_name();

// Directly access via name
AName.field_name = value;
AName.command_name();

// Indirectly access via a variable
// containing either a name or a handle
%AVar.field_name = value;
%AVar.command _name();
```

To get a picture of how this works for real do this:

1. Start the SDK
2. Run one of the Missions
3. Start the World Editor Inspector
4. Switch to camera view and select the character
5. Give the character a name: *egt_dude*
6. Remember the character's handle (will vary) ####
7. Open the console (~)
8. type:

```
$player_id = ####;
$player_name = ####.getname();

echo($player_id.position);
echo($player_name.getid());
echo("egt_dude".getid());
echo(egt_dude.getid());
```

Dynamic Fields

In addition to normal fields, which are common between all instances of an object type, Torque Script allows you to create dynamic fields. Dynamic fields are associated with a single instance of an object and can be added and removed at will. You've already learned how to do this with the Inspector. Adding a dynamic field in Torque Script is automatic. If you reference a field in the context of an object and the field is not found, it will be created.

```
// new_var will be created and intialized to 0
echo($player_id.new_var);

// new_var2 will be created and intialized to "Hello"
$player_id.new_var2 = "Hello";
echo($player_id.new_var2);
```

Console Methods

In addition to supporting the creation of functions, Torque Script allows you to create methods which have no associated C++ counterpart:

```
function Classname::method_name(%this, [arg0],...,[argn]) {
    statements;
    [return val;]
}
```

The syntax breaks down as follows:

- **function** – Is a keyword telling Torque Script we are defining a new function.

- **ClassName**:: – Is the class type this function is supposed to work with.
- **func_name** – Is the name of the function we are creating.
- **%this** – Is a variable that will contain the handle of the 'calling object'.
- … – Is any number of additional arguments.

At a minimum, Console Methods require that you pass them an object handle. You will often see the first argument named %this. People use this as a hint, but you can name it anything you want. As with console functions any number of additional arguments can be specified, separated by commas. Also, a console method may return an optional value.

Here are some examples:

```
function Goober::hi(%this) {
echo("Goober Hello ", %this);
}
```

Assuming our player handle is 1000, if we type:

```
1000.hi();
```

we get,

```
<input> (0): Unknown command hi.
Object (1000) Player->ShapeBase->GameBase
->SceneObject->NetObject->SimObject
```

What has happened is that Torque has searched the entire hierarchy of Player and its parent classes, looking for a function called hi() defined in the context of one of those classes. Not finding one, it prints the above message. To demonstrate that Torque does search the class hierarchy of Player, try this next:

```
function NetObject::hi(%this) {
echo("NetObject Hello ", %this);
}
```

typing,

```
1000.hi();
```

we get,

```
NetObject Hello 1000
```

Next, if we define:

```
function Player::hi(%this) {
echo("Player Hello ", %this);
```

```
Parent::hi(%this);
}
```

we would type,

```
1000.hi();
```

and get,

```
Player Hello 1000
NetObject Hello 1000
```

Do you see what happened? Torque found Player:hi() first, but we also wanted to execute the next previous defintion of hi(). To do this we used the '*Parent::*' keyword. Of course, not finding a ShapeBase instance, which is Player's literal parent, Torque then searched down the chain till it came to the NetObject version. Vioala!

Lastly, we can force Torque to call a specific instance as follows:

```
NetObject::hi(1000);
```

gives us,

```
NetObject Hello 1000
```

and

```
ShapeBase::hi(1000);
```

also gives us,

```
NetObject Hello 1000
```

since there is no ShapeBase instance of hi() defined. Too cool.

Things to know:

- Defining subsequent console methods with the same name as prior console methods, over-rides the previous definition permanently unless the re-definition is within a package (see packages below).

Packages

Packages provide dynamic function-polymorphism in Torque Script. In short, a function defined in a package will over-ride the prior definition of a same named function when the package is activated. Packages have the following syntax:

```
package package_name() {
function function_defition0() {
}
...
function function_defitionn() {
}
};
```

Things to know:

- The same function can be defined in multiple packages.
- Only functions can be packaged.
- Datablocks (see below) cannot be packaged.

Packages are can be activated,

```
ActivatePackage(package_name);
```

and deactivated:

```
DeactivatePackage(package_name);
```

The easiest way to get a feel for packages is with a quick example. Assuming that you have installed the EGT Lesson Kit (if not, see EFM), you can load the following code from disk by typing this in the console:
exec(egt_base/samples/tech_scripting_packages01.cs");

```
//
// Define an initial function: demo()
//
function demo() {
    echo("Demo definition 0");
}

//
// Now define three packages, each implementing
// a new instance of: demo()
//
package DemoPackage1 {
function demo() {
    echo("Demo definition 1");
}
};
package DemoPackage2 {
function demo() {
    echo("Demo definition 2");
}
};
package DemoPackage3 {
function demo() {
    echo("Demo definition 3");
    echo("Prior demo definition was=>");
    Parent::demo();
}
};

//
// Finally, define some tests functions
//
function test_packages(%test_num) {
    switch(%test_num) {
    // Standard usage
    case 0:
        echo("----------------------------------------");
        echo("A packaged function over-rides a prior");
        echo("defintion of the function, but allows");
        echo("the new definition to be \'popped\' ");
        echo("off the stack.");
        echo("----------------------------------------");
        demo();
        ActivatePackage(DemoPackage1);
        demo();
        ActivatePackage(DemoPackage2);
        demo();
        DeactivatePackage(DemoPackage2);
        demo();
        DeactivatePackage(DemoPackage1);
        demo();
```

WARNING: Code is continued on next page (temporary fix for PDF bug that killed remainder of table).

© Hall Of Worlds, LLC. All rights reserved.

```
    // Parents
    case 1:
        echo("----------------------------------------");
        echo("The Parent for a packaged function is");
        echo("always the previously activated ");
        echo("packged function.");
        echo("----------------------------------------");
        demo();
        ActivatePackage(DemoPackage1);
        demo();
        ActivatePackage(DemoPackage3);
        demo();
        DeactivatePackage(DemoPackage3);
        DeactivatePackage(DemoPackage1);
        echo("----------------------------------------");

        demo();
        ActivatePackage(DemoPackage1);
        demo();
        ActivatePackage(DemoPackage2);
        demo();
        ActivatePackage(DemoPackage3);
        demo();
        DeactivatePackage(DemoPackage3);
        DeactivatePackage(DemoPackage2);
        DeactivatePackage(DemoPackage1);
    // Stacking oddities
    case 2:
        echo("----------------------------------------");
        echo("Deactivating a \'tween\' package will");
        echo("deactivate all packages \'stacked\' after");
        echo("it.");
        echo("----------------------------------------");
        demo();
        ActivatePackage(DemoPackage1);
        demo();
        ActivatePackage(DemoPackage2);
        demo();
        DeactivatePackage(DemoPackage1);
        demo();
    }
}
```

The standard way to use a packages is to define a previously defined function inside the package, activate it as needed, and then deactivate it to go back to the 'default' case for the function. To see this in action, type: **`test_packages(0);`**

Torque script provides a useful feature, the 'Parent'. By using the '*Parent::*' keyword in a packaged function may execute the function it is over-riding. To see this in action, type: **`test_packages(1);`**

It is important to understand that packages are (for all intensive purposes) stacked atop each other. So, if you deactivate a package that was activated prior to other packages, you are in effect automatically deactivating all packages that were activated prior to it. To see this in action, type: **`test_packages(2);`**

Things to know:

- Deactivating packages activated prior to other packages deactivates all prior active packages.
- Packages may define new functions. Remember, when you deactivate a package, these functions get removed from the namespace.
- *Parent::* keyword is only valid in the scope of a packaged function.
- *Parent::* keyword is not recursive:
    - i.e. Parent::Parent::fun() is illegal.

Namespaces

As previously mentioned, namespaces are provided in Torque Script. They way they work is quite simple. First, all objects belong to a namespace. The namespace they belong to normally defaults to the same name as the class.

```
// Player class Namespace
Player::
```

Also as previously mentioned, these namespaces provide separation of functionality, such that one may have functions with the same name, but belonging to separate namespaces. To use one of these functions, you must either manually select the appropriate namespace, or in some cases this is done automatically for you. I'll re-address this below when we discuss console methods.

It is important to understand that the "**::**" is not magical in any way. In fact, you can create functions with "**::**" in their name. This doesn't mean they belong to a namespace. If the expression prefixing the "**::**" is not a valid class/namespace name, in effect, all you have done is create a unique name.

```
// Not really  namespaces
function Ver1::doit() {
...
};

function Ver2::doit() {
...
};
```

We will discuss the creation of new functions and methods in an objects namespace below.

Lastly, there is a way to create new non-class namespaces. This discussion will be deferred to the advanced topics section of this chapter.

Datablocks

Of all the features in Torque Script, *Datablocks* are probably the most confusing. To make things worse, they are central to the creation of most objects, which means you need to understand them relatively early. I will give a summary of datablocks here, but because you need to understand some other more advanced topics prior to really jumping into *datablocks* I will defer the in-depth review till later (see 'Datablocks Revisited' below).

The official definition of *datablock* is:

> **"Datablocks are special objects that are used to transmit static data from server to client."**
> – engine.overview.txt

In the words of Keanu Reeves himself, "Whoaa…". Or perhaps in my own words, "huh?". This definition, although true, didn't really tell me much. Some searching turned up additional definitions:

> **"A datablock is an object that contains a set of characteristics which describe some other type of object."**
> – Joel Baxter

"Better, but I'm still a little blurry on the purpose and use of datablocks."

> **"A datablock is a(n) object that can be declared either in C++ engine code, or in script code ... Each declared datablock can then be used as a "template" to create objects..."**
> – Liquid Creations, Scripting Tutorial #2

"OK, now I get it. Datablocks are templates and we use them to create new objects with the attributes specified by the template. Cool." But, how do we do this? For the answer to that question, you'll have to wait. First we need to discuss a few other important topics, then we will revisit *datablocks* and give them the thorough coverage that they deserve.

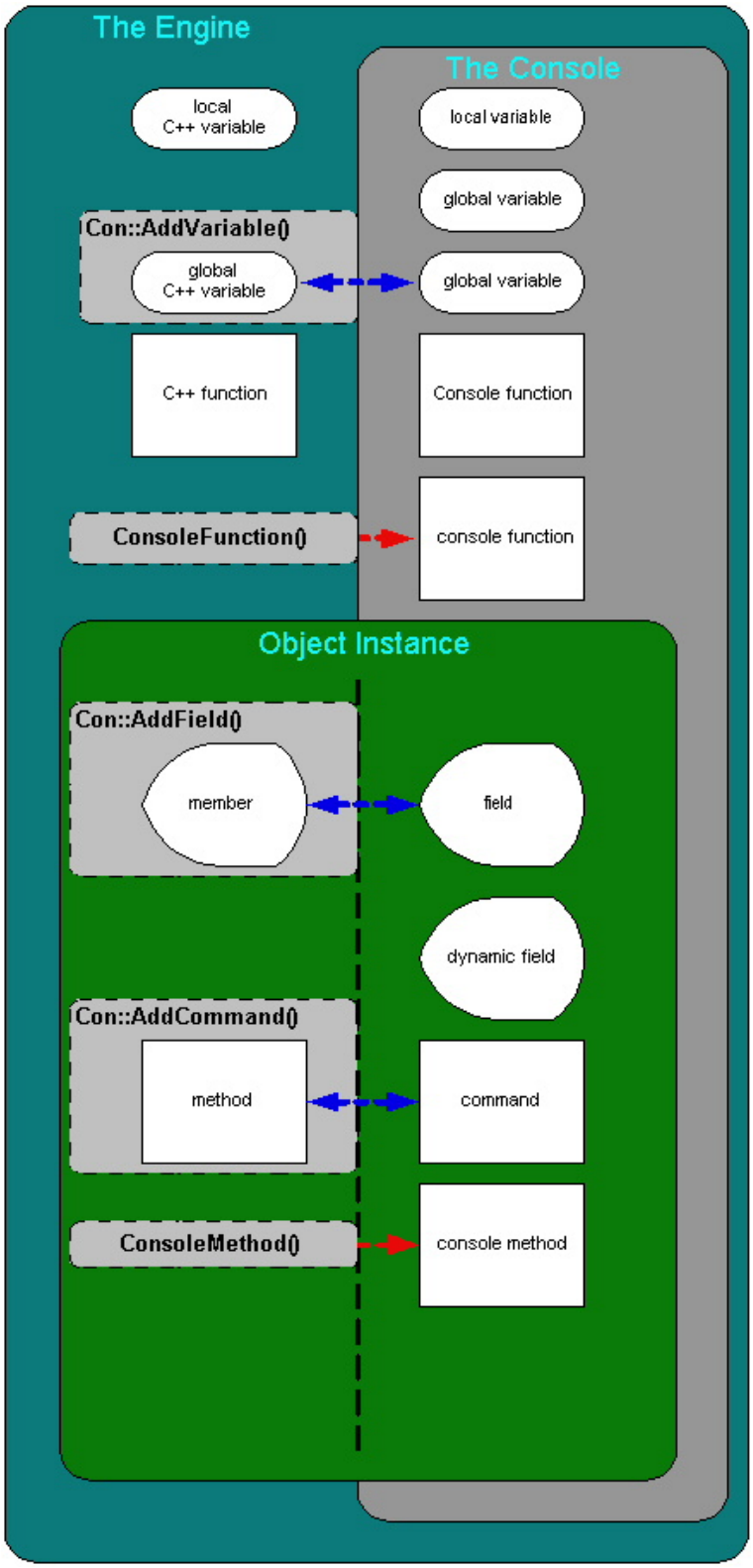
Interfacing with the Engine

All right, up until this point, we have talked about scripting alone. Now we will discuss the interface between the console and the core engine.

Before starting it is important that we define a consistent set of terms to describe the concepts we are about to explore:

Term	Definition
C++ Variable	• A C++ variable not associated with a class.
C++ Function	• A routine defined and declared in C++ not associated with a class.
Member	• A variable defined and declared in a C++ class. • Can only be accessed in association with an instance of an object.
Method	• A routine defined and declared in a C++ class. • Can only be accessed in association with an instance of an object.
Local (Variable) or Global (Variable)	• A variable in the Console. • Global variables may be defined in C++ an linked to a global engine variable. Allowed C++ modifiers: const, static.
Console Function	• A routine in the Console, not associated with any particular namespace. • May be defined entirely in script or C++.
Field	• A variable associated with an object in the Console. • Linked with a Member.
Dynamic Field	• A variable associated with an object in the Console. • Exists only in context of Console.
Command (Deprecated)	• A routine associated with a particular namespace in the Console. • Linked with an existing Method.
Console Method	• A routine associated with a particular namespace in the Console. • Exists only in context of Console.

As they say, “a picture is worth a thousand words”. So, in that vein, here is an illustration of the above table:



© Hall Of Worlds, LLC. All rights reserved.

Engine Interfacing Problem-Solution Matrix

Before we jump into the gory details of interfacing the engine and console, here is a problem solution matrix:

Problem (Want to…)	Solution	Solution Type
C++ to Console		
Expose **Member** as **Field**.	addField() addFieldV()	FUNCTION
Expose **Member** as **Field**.	addNamedField() addNamedFieldV()	MACRO
Expose/Remove global **C++ Variable** or static **Member** as **Local Variable**	Con::addVariable() Con::removeVariable()	FUNCTION
Expose **Method** as **Command**.	Con::addCommand() (**Deprecated**)	FUNCTION
Create **Console Method** from C++.	ConsoleMethod()	MACRO
Create **Console Function** from C++.	ConsoleFunction()	MACRO

addField()

This function allows you to link C++ class members to console object fields.

The rules for using this function are:

1. Member to be exposed must be non-dynamic (i.e. not a pointer).
2. Member to be exposed must be non-static.
3. addField() statements must be included in the class initPersistFields() method.

```
void ConsoleObject::addField(const char*  in_pFieldname,
                        const U32 in_fieldType,
                        const dsize_t in_fieldOffset,
                        const char* in_pFieldDocs)

void ConsoleObject::addField(const char*  in_pFieldname,
                        const U32 in_fieldType,
                        const dsize_t in_fieldOffset,
                        const U32 in_elementCount,
                        EnumTable *in_table,
                        const char* in_pFieldDocs)
```

- **in_pFieldname** – String specifying variable name as used in console.
- **in_fieldType** – The variable type. (types specified in consoleTypes.h; see Types Appendix EFM).
- **in_fieldOffset** – This is a numeric value calculated using the Offset() macro (see below).
- **in_elementCount** – Numer of elements at offset. The default value is 1, but if you are referencing an array then this value will be the number of elements in the array.
- **in_table** – This last argument is used when the member type is *TypeEnum*. In this special case, you need to define an *EnumTable* containing a map of the ENUM values and the strings to represent them in the console. (see below EFM).
- **in_pFieldDocs** – EFM

Here is an example (found in guiCrossHairHud.cc) :

```
class GuiCrossHairHud : public GuiBitmapCtrl
{
...
   ColorF   mDamageFillColor; // declared here
...
void GuiCrossHairHud::initPersistFields()
{
..
   addField( "damageFillColor", TypeColorF, Offset( mDamageFillColor,
              GuiCrossHairHud)); // added in initPersistFields()
```

In this code, the member *mDamageFillColor* has been linked to the field *damageFillColor*. If you wish to test your ability to inspect and modify this field, do the following:

- Start the SDK
- Run any mission
- Open the GUI Editor (F10) and find the GuiCrossHairHud in the GUI Inspector Tree.
- Remember the control's handle.
- Stop the GUI Editor (F10)
- Open the console (~)
- try these commands (substitute handle for ####):

```
echo(####.damageFillColor);
####.damageFillColor = " 1.0 0.0 0.0 1.0";
echo(####.damageFillColor);
```

You may notice that although the contents of the variable were changed (to 100% opaque RED), the reticle stayed GREEN. This is because you changed the Server-copy of the variable, not the Client-copy. I will elaborate on this in the Networking chapter.

For clarity's sake here are some made up samples demonstrating the use of this function:

```
//// Adding simple variable

// the varible we want to add
bool   bTorqueRocks;
// adding it
addField( "TorqueRocks", TypeBool, Offset(bTorqueRocks,EGTClass));


//// Adding an array
// the varible we want to add
S32   arynEGTChapters[28];
// adding it
addField( "TorqueRocks", TypeS32, Offset(arynEGTChapters,EGTClass),28);


//// Adding an ENUM
// the varible we want to add
StateData::LoadedState stateLoaded[MaxStates];
// the EnumTable
static EnumTable::Enums enumLoadedStates[] =
{
   { ShapeBaseImageData::StateData::IgnoreLoaded, "Ignore" },
   { ShapeBaseImageData::StateData::Loaded,       "Loaded" },
   { ShapeBaseImageData::StateData::NotLoaded,    "Empty" },
};
static EnumTable EnumLoadedState(3, &enumLoadedStates[0]);
// adding it
addField("stateLoadedFlag", TypeEnum, Offset(stateLoaded,
ShapeBaseImageData), MaxStates, &EnumLoadedState);
```

Having covered the most commonly used version of the addField() functions, lets quickly outline the uses and syntax for the other varieties.

addFieldV()

This is a specialized version of the addField() function. It does not handle arrays or ENUMs, but it has the nice feature of a validator function. Let us look at the syntax and then I'll explain validator functions.

```
void ConsoleObject::addFieldV(const char*  in_pFieldname,
                              const U32 in_fieldType,
                              const dsize_t in_fieldOffset,
                              TypeValidator *v)
```

- **in_pFieldname** – String specifying variable name as used in console.
- **in_fieldType** – The variable type. (types specified in consoleTypes.h; see Types Appendix EFM).
- **in_fieldOffset** – This is a numeric value calculated using the Offset() macro (see below).
- ***v** – This is a pointer to a TypeValidator class of which there are several derived types to choose from.

Validator Functions

It is easiest to explain *TypeValidators* with some examples:

```
EFM
```

addNamedField() and addNamedFieldV()

These two varients of addField() are actually MACROS that do nothing more than give the console field the same name as the class member which is being accessed.

```
#define addNamedField(fieldName,type,className) \
    addField(#fieldName, type, Offset(fieldName,className))
```

```
#define addNamedFieldV(fieldName,type,className, validator) \
    addFieldV(#fieldName, type, Offset(fieldName,className), validator)
```

Offset() MACRO

I glossed over the Offset macro above in the interest of first discussing the usage of addField() and its many versions. The offset macro is a cool bit of coding that calculates the position of a variable within an instance of a class. Here is the syntax:

```
Offset(VariableName, ClassName)
```

- **VariableName** – This is the (C++) name of the class member we want to expose.
- **ClassName** – This is the name of the class the member resides in.

```
// Coolness ==> Offset Macro Definition:

// GCC Version
#define Offset(m,T) ((int)(&((T *)1)->m) - 1)

// Default Version
#define Offset(x, cls) ((dsize_t)((const char *)\
 &(((cls *)0)->x)-(const char *)0))
```

removeField()

This function allows you to unlink a previously linked member-field pair. i.e. This removes the field from the console. Simple as that. Why doit? Consider the case where you derive from an object that does an addField() call for a member that you have decided should not be accessible (like position for Terrain data…).

```
bool ConsoleObject::removeField(const char* in_pFieldname)
```

- **in_pFieldname** – String specifying field to be removed.

```
// 1. TerrainBlock is inherited from SceneObject.
// 2. SceneObject links member mObjToWorld with position.
// 3. TerrainBlock undoes this(in terrData.cc)
removeField("position");
```

Con::addVariable

This function allows you to expose a global C++ variable or a static Member as a global variable in the console. see: camera.cc

```
Con::addVariable(const char *name, S32 t, void *dp);
```

- **name** – String specifying name of global variable (in console).
- **t** – The variable type. (types specified in consoleTypes.h; see Types Appendix EFM).
- **dp** – A pointer to the global C++ variable, or static Member.

```
// From camera.cc
Con::addVariable("Camera::movementSpeed",TypeF32,&mMovementSpeed);)
```

The above code exposes the global C++ variable mMovementSpeed in the console as a global variable named Camera::movementSpeed.

Note: This controls the free camera flight speed while editting. Now you know another way to set it beyond the normal editor keystrokes.

Con::removeVariable (deprecated)

This function allows you to remove a global variable from the console that was previously added with one of the variants of the addVariable() function.

```
bool removeVariable(const char *name)
```

- name – String specifying name of global variable (in console).

Upon searching, I found not a single instance in which this was used. This is probably because it would be a sign of poor planning to add then remove a variable. Having the source code, you could just remove the add call and be done with. Nonetheless, you can use this function if it suits you to do so.

Con::addCommand (deprecated)

This function allows you to expose a **Method** as a **Console Method**. However, it is now deprecated. i.e. You are not encouraged to use it. Instead, you should create a method in C++ (if you need to use it in the engine), and then call this method from within a ConsoleMethod (see below).

ConsoleMethod

This is a macro which allows you to create a new **Console Method** from C++. The use of ConsoleMethod() is not much expanded in the engine as it has been chosen as a means of replacing the more clumsy addCommand() calls. The 'static' is used when you want a want to create a console method that accesses/uses static class members.

```
ConsoleMethod(className,scriptname,returnType,minArgs,maxArgs,usage)
ConsoleStaticMethod(className,scriptname,returnType,minArgs,maxArgs,usage)
```

- **className** – Is the name of the class the method is in.
- **scriptname** – Is the name the method will be given in the console (i.e. used by Torque Script).
- **returntype –** Is the return type of the method.
- **minargs –** Is the minimum arguments this method takes.
  - **Note:** 2 Is the minimum, because the name of the console method is automatically passed as the first argument and the handle is automatically passed as the second argument.
- **maxargs –** Is the maximum number of args that can be passed to this method.
  - **Note2:** If you put 0 in this field, it means any number of arguments may be passed to the method.
- **usage –** Is a string that will be printed as a help statement if someone later attempts to use this method with the wrong number of arguments.

```
//From SimBase.cc

ConsoleMethod(SimObject, getId, S32, 2, 2, "obj.getId()")
{
   argc; argv;
   return object->getId();
}
```

In the above function, we've (well, the engine authors really…) written a simple utility to return the current object's unique ID (its handle). A short breakdown is as follow:

- It can be called on (by) all objects which are SimObjects or children of SimObject
- The name of the function in Torque Script will be 'getId' or more specifically 'SimObject::getId'.
- 'getId' takes two arguments, the ConsoleMethod name and the objects handle (which by the way is the same thing we're being asked to return).
- The ConsoleMethod is expected to return a signed 32-bit value.
- The ConsoleMethod will take a minimum and a maximum of two arguments.
- The usage message is: "obj.getId()"
- Internally,
  - we do nothing with the standard argc and argv variables.
  - we call a method named 'getId()' to return a signed 32-bit value.

ConsoleFunction

This is a macro which allows you to create a new **Console Function** from C++. Ex: ShapeBase.cc

```
ConsoleFunction(name,returnType,minArgs,maxArgs,usage)
```

- **name** – This is the name of the function as it will be used in the console.
- **returnType** – Is the return type of the function.
- **minArgs** – Minimum arguments this function can accept.
  - **Note:** 1 Is the minimum, because the name of the function is automatically passed as the first argument.
- **maxArgs** – Maximum arguments this function can accept.
  - **Note2:** If you put 0 in this field, it means any number of arguments may be passed to the function.
- **usage** – Is a string that will be printed as a help statement if someone later attempts to use this function with the wrong number of arguments.

```
// From main.cc
ConsoleFunction( getSimTime, S32, 1, 1, "getSimTime() - Time since game
started.")
{
   return Sim::getCurrentTime();
}
```

First, I corrected the above consolefunction declaration for this example. In the engine, the usage parameter was used instead to give information about the function's purpose. Really, we'd want it to tell us what to do if we messed up the args list. Its pretty clear that this creates a function in the console named 'getSimtTime', which returns a signed 32-bit value, and takes 1 argument. Interestingly, internally, we call the static method 'Sim::getCurrentTime()'. Remember, console functions can internally call any function in the scope of the macros declaration or any static method.

Additional Engine Interfacing Functions

In addition to the major functions we've discussed so far, there are some other important, though less used, functions you should know about. I will cover them briefly, and rely on your ability to examine the source code for usage examples. Note: I will use some of these in the tutorials that come with this guide also.

Con::getLocalVariable()

This function allows you to get the contents of a local variable in the console from within the engine. Note: The value is returned as a char string.

```
const char *Con::getLocalVariable(const char* name);
```

- **name** – This is the name of the local variable in the console.

Con::setLocalVariable()

This function allows you to set the contents of a local variable in the console from within the engine. Note, all values passed as char strings to the console. The console automatically converts this information as necessary.

```
Con::setLocalVariable(const char *name, const char *value);
```

- **name** – This is the name of the local variable in the console.
- **value** – This is the new value to place in the console variable.

Con::printf()

Con::warnf()

Con::errorf()

These functions provide the ability to print various levels of information into the console (and subsequently the log if logging is enabled).

```
Con::printf(const char *_format, ...);
```

- **_format** – A standard C printf formatting string.
- **…** – Any number of arguments associated with the formatting string contents

```
Con::warnf(ConsoleLogEntry::Type type, const char *_format, ...);
```

- **type** – A category indicator, where the category can be:
    - General
    - Assert
    - Script
    - GUI
    - Network
- **format** – A standard C printf formatting string.
- **…** – Any number of arguments associated with the formatting string contents

```
Con::errorf(ConsoleLogEntry::Type type, const char *_format, ...);
```

- **type** – A category indicator, where the category can be:
    - General
    - Assert
    - Script
    - GUI
    - Network
- **format** – A standard C printf formatting string.
- **…** – Any number of arguments associated with the formatting string contents

© Hall Of Worlds, LLC. All rights reserved.

Datablocks Revisited

As previosly promised, we will now go into greater depth on the usage and purpose of datablocks.

datablocks discussion
http://www.garagegames.com/index.php?sec=mg&mod=forums&page=result.forum&qtm=datablock

EFM

Special Topics

Network Scripting

For a more detailed coverage of network scripting go to the networking chapter of this guide.

Common Scripting Tasks

Instead of presenting a list of common scripting problems and solutions in this chapter, I have dedicated an appendix of the same name to this purpose.

Debugging Scripts

As with the other special topics, this will not be covered here. Instead, please go to the debugging chapter.

EFM – Add section detailing scripting tasks and section giving usage examples for script available functions (as much as possible):
consolefunctions.cc
mathtypes.cc
sceneobject.cc
initContainerRadiusSearch and other goodies
shapebase.cc – pointInWater, oncollision, ???