Basic Lessons in Writing Scripts for Blitzkrieg 1

by hitandrun, June 2006

Debugging notes by BKP

Special thanks to Wespex and those who proof-read and made suggestions.

Introduction

When the game Blitzkrieg was released, the Map and Resource Editors were bonus features packaged with the game. The Help Files for the editors were limited and there were absolutely no instructions for writing the code (**script**) which would control events in custom missions. Working out how to get the most out of the editors was a cumulative process conducted by players sharing their discoveries on various forums. Unpicking the script language, Lua, was similarly achieved by good code breaking, trial and error and a lot of very open sharing of discoveries. The numerous individuals who took on the task of "cracking" Lua have made custom map creation a much easier prospect for all of us who follow. While those pioneers are not always active in BK anymore, everyone who plays custom maps or tries to make them owes their ability to do so to a lot of hard work by some determined code busters who were generous enough to share their discoveries with the rest of the BK community.

One of the most important things to come out of the code busting was Calvin's *Blitzkrieg Guide to Programming Lua Functions*. This made available to everyone a list of the script calls used in BK with a guide to how they are used. As this is a "must have" for anyone writing BK scripts IT SHOULD BE downloaded. The Guide is like a dictionary of Lua but it does not provide easy instructions for how the terms are combined into a working script (Calvin includes an example but it could hardly be described as "introductory" and is best skipped over by anyone new to scriptwriting). Calvin's *Blitzkrieg Guide to Programming Lua Functions* can be downloaded from Blitzkrieg Portal

Wespex was the first to produce an integrated tutorial for making a map, writing the script and creating a "wrapped up" game which could be shared with other players. That tutorial (republished several times and finally in combination with Calvin's Guide) is the primary introduction for anyone wanting to make custom maps. Following publication of the tutorial, Wespex provided a very comprehensive series of answers to scripting questions on BKP Forum. The most valuable characteristic of both the Tutorial and the Answers was the comprehensive notes provided in the script examples which not only provided a ready-made fix but explained how and why the script worked. It is probably fair to say that the vast majority of people currently writing scripts learnt by following the Tutorial.

These Basic Lessons have been written in response to a number of comments on the Blitzkrieg Portal forums and the continuing difficulties apparently experienced by many mapmakers when they try to start writing scripts. The Basic Lessons attempt to demonstrate the simplest principles of Lua and then show how to build up a script to a point which could easily provide the basis for a custom game. It will attempt to do this while demystifying some technical terms.

Points to Consider

As someone who has only recently started writing Lua I am very familiar with what a big hurdle getting started is and how frustrating it is when scripts don't work. However, there are some points to bear in mind:

Lua is a language and you must learn what the words mean and how to use them from scratch. Just because you are very clever doesn't mean you should expect to be able to write Lua without doing some hard work to learn the vocabulary and grammar

Unlike human languages, if you make a *tiny* mistake in Lua then you will be *completely* unintelligible. Most frustration when starting to write Lua comes from scripts which "just don't work". Almost inevitably it's your fault! The natural reaction is to say "the program doesn't work" or "there's a problem with my computer" but 999/1000 times you have made a typing error or made a grammar mistake.

Making your own map and seeing your own script run is great fun; *getting* the script to run isn't always fun. The very best scriptwriters in the BK community all admit that they still spend a lot of frustrating time fixing their own creations. Not everyone will have the patience to go through there scripts line by line looking for one letter out of place, one bracket missing or one extra semicolon. If you want to make scripts you have to accept that that is what is involved and no one else can make it any easier for you.

Once you have cracked the basics of Lua, the best way to learn is to open up a mission by a mapmaker you like and go through the script line by line (with a copy of Calvin's Guide at your side) working out what is going on.

If you want to try something new, or use something you have "borrowed" from another scriptwriter, don't try it in the middle of you pride-and-joy 26x26 map. Make a test map 4x4 with only the minimum number of units and try out the individual block of script which is new for you. That way if it doesn't work you can spot it quickly and not have to check a lot of other, unrelated stuff to see if it is running interference. This is the method used in the lessons which follow.

If you are really stuck (after going through BKP's **Debugging Notes** accompanying these lessons) then search the Forum, most problems have been addressed before. If you can't find what you are looking for then post a question under Map Making Questions and Answers. If the question is specific (perhaps copy and paste a bit of the script) then you are likely to find you get a quick and helpful reply from one of the gods of scriptwriting who so generously help out us mere mortals.

How the Basic Lessons Work

The following **Lessons** each refer to the map (.bzm) and script (.lua) file of the same name that are included in the archive with this document. All the .bzm and .lua files should be copied and pasted into the Blitzkrieg\Run\Data\Maps folder on your computer; this is also where you will need to save your own custom maps and scripts. For each lesson, open the map in MapEditor and the script in SciTE editor. Once you have looked at both and read the lesson, then press the Run Blitzkrieg button on the MapEditor toolbar (top right) to see the script execute.

Each of the first five lessons build on the previous one so that the map and script can be seen developing. The last two lessons are separate and cover topics which often come up in BKP Forum questions.

These lessons assume that you have downloaded the Blitzkrieg SciTE Editor. As it is very helpful, free and not very big to down load from BKP downloads page there really is no reason not to use it. It won't write the script for you, it won't find your mistakes or tell you when a script won't run, but it is very helpful when writing and checking scripts. **So download it now!**

Blitzkrieg SciTE Editor

Note: when using the SciTE Editor you must give the filename a .lua suffix otherwise it will be saved as .xml.

Basic Principles

The *script* is written as a .lua file. This is *linked* to the map in the MapEditor using the button that looks like a spanner on a letter A.

The script contains a series of instructions, called **functions**, which make things happen when the map is run. Functions can be very simple, making just one thing happen at a specified time, or more complex, testing a condition and then making different things happen depending on the outcome.

Note: A list of Blitzkrieg-specific functions can be found in **Appendix A**

In order for functions to interact with the map units, squads and buildings must be given a **ScriptID** in the map editor. With the Objects tab selected, double click on a unit to call up the unit Properties window then double click on the ScriptID box and type in an ID, then close the Properties window.

Tip: Make a note of which units have which ScriptID. Some mapmakers include the key to their numbering at the start of their script (see Inserting Notes).

Tip: It is often helpful to give the player's units and the AI's units clearly different ScriptID numbers, e.g.: 100s and 200s).

Many functions make use of **ScriptAreas**. These are defined in the MapEditor using the Map Tools tab: select the Script Area Tool and drag the cursor over part of the map to define an area, and then type in the name of the area. When writing the script you must use the name of the ScriptArea *exactly* as it is typed in the MapEditor (case and space sensitive).

Tip: Give your ScriptAreas distinct and meaningful names so that you do not mix them up in the script (Farm and Town NOT area1 and area2).

Lua Language

The syntax for Lua is simple and consistent and *must* be followed exactly. Any spelling, spacing or punctuation mistake will result in the script not running or failing to function correctly. The majority of script problems are typing and syntax errors.

The following words have special meanings in Lua and are known as **keywords**:

and	break	do	else	elseif	
if	end	false	for	function	
in	local	nil	not	or	
repeat	return	then	true	until	while

It does not matter what you think these words mean in the "real world", in Lua they have fixed uses, hardwired into the language, and they can *only* be used in that way. The use of some of them will be explained in these lessons. By convention I use these keywords in lowercase.

There are also some Blitzkrieg keywords, like >RunScript<, >LandReinforcement<, >Win< and >Loose<. These have to be written with capital and lowercase letters in exactly the right places. Calvin's guide is the dictionary for how to write these correctly.

Tip: if you use the Blitzkrieg SciTE editor then all keywords will appear in blue with a pale blue background when they have been typed correctly.

The names of things which the scriptwriter defines are called **names**. These can have any numbers and letters in them but *must not* start with a number. By convention I start names with a capital letter. For example, in Lesson1 I write

function MoveCar()

where >function< is a keyword and >MoveCar< is a name.

Tip: if you use the Blitzkrieg SciTE editor then names will appear in black type

Once a name has been defined as a function it can be *called* later in the script, but when it is used again as a **string** it must be put in quotation marks. For example in Lesson1 I go on to write

RunScript("MoveCar", 3000)

where >RunScript< is a Blitzkrieg keyword and >"MoveCar"< is a string which refers to a function which I have already *named* elsewhere in the script.

It is absolutely essential that the strings be typed *exactly* as they occur in the function name or else the script will not recognize them. For that reason it is often best to copy and paste names and strings so that typing errors do not creep in. When using the quotation marks there must not be any space between the mark and the beginning or end of the string.

*Tip: if you use the Blitzkrieg SciTE editor then strings will appear in purple text when the quotation marks are in the right places. If you see purple background it means that one set of quote marks is missing. Note that SciTE editor **does not** check that you have typed the string correctly!*

Functions and keywords are *qualified* by **parameters**. To understand parameters think of the function or keyword as someone shouting "Do!" at you. This does not have any meaning until you have be told something else such as Do What, Do It When, Do It How Often: these qualifying terms are the parameters for the function Do.

In Lua parameters are always specified in pairs of brackets. In the example

RunScript("MoveCar", 3000)

the keyword >RunScript< needs to know which script to run and when, these two parameters are then defined inside the brackets as run the string >"MoveCar" < and the **value** 3000 (this is further explained in the lesson below).

Tip: if you use the Blitzkrieg SciTE editor then values will be shown in pale blue type.

Where more than one parameter is given the strings or values are separated by a single comma. This is very important and the script will not run if the comma is missing.

Some keywords and functions are not defined by any parameter, that is to say they are self-explanatory for the script and do not need to be qualified by anything such as what, where, when or how often. Where a function or keyword has no parameters the brackets are left empty, *but* the brackets must still be used. Common examples include >Suicide()< and the empty brackets after a function name. Missing an **empty bracket** is a very common reason for a script not to run.

Indentations are commonly used by scriptwriters but are not necessary and do not effect the script. They are a very helpful way of organizing things and I will use my own approach in the examples. Other scriptwriters have their own approaches.

By convention every line of script ends with a **semicolon** (except those where a keyword will operate on the following line as well – covered in Lesson3 below).

Notes are sometimes included in scripts to help the scriptwriter keep track of things or to show others what is going on. A note is started and finished by at least two (2) dashes: -- NOTE --. A note is not *read* by the script. Therefore putting two dashes at the start and end of a line can effectively disable that line. This can be useful during testing.

Tip: if you use the Blitzkrieg SciTE editor then notes will appear in green text.

Lesson 1 – Writing a function and making it run

This lesson will make one unit move from its starting position to a defined point

Lesson1 Map

There is a car with Script ID >100< (double lick on unit to see ID) and building which the car will drive to.

Lesson1.lua Script

```
function MoveCar()
        Cmd(0, 100, 916, 1516);
        Suicide();
end;

function Init()
        RunScript("MoveCar", 3000);
end;
```

NOTE: the executable scripts are packed as .lua files and you can copy and paste from them to create you own scripts. Never try to write scripts in Word which embeds text codes which will stop a lua file from working.

Explanation

function MoveCar()
establishes a function named >MoveCar< which can be *called* by the script

Cmd(0, 100, 916, 1516);
gives the order Move (0) to the unit (ID 100), to go to position x,y, (916 East, 1516 North)

Suicide();
Tells the script not to keep running this function

end;
closes an active part of the script, in this case >function MoveCar<

function Init()
this is the START function, every script must have one **function Init** or else it cannot get going

RunScript("MoveCar", 3000);
does what it says! It runs the script called >MoveCar< (do not forget the quote marks) and says when it should run (3000 milliseconds which is 3 seconds after the game starts)
end;
closes an active part of the script, in this case >function Init<

Note: a list of the different Cmd codes and how to use them is in Calvin's guide.

Lesson 2 – Adding a second function with multiple commands

This lesson will make a second unit perform a series of moves at a different time from the first unit.

Lesson2 Map

There is a car with ScriptID >100<, a truck with ScriptID >101<, and there is a tree and a monument which the truck will drive around on its way to the building.

Lesson1.lua Script

```
function MoveCar()
        Cmd(0, 100, 916, 1516);
        Suicide();
end;

function MoveTruck()
        Cmd(0, 101, 1583, 223);
        QCmd(0, 101, 1890, 1570);
        QCmd(0, 101, 1100, 1890);
        Suicide();
end;

function Init()
        RunScript("MoveCar", 3000);
        RunScript("MoveTruck", 7000);
end;
```

Explanation

function MoveCar()

Cmd(0, 100, 916, 1516);

Suicide();

end;

function MoveTruck()

a new function defined just like the first but with a new name

Cmd(0, 101, 1583, 223);

the order Move (0) is the same as for the car but the ScriptID of the truck is different (101) as are the coordinates

QCmd(0, 101, 1890, 1570);

QCmd is just like Cmd except that it happens after the pervious command has run. This command relates to the same truck (101) but now sends it to a second set of coordinates

QCmd(0, 101, 1100, 1890);

Suicide();

```
end;

function Init()
        RunScript("MoveCar", 3000);
        RunScript("MoveTruck", 7000);
```
the instruction to run the new function is added to the *start* (Init) function but this time it runs 7 seconds after the game begins
```
end;
```

Note

The second function could also be started *in* the first one as shown below. This is often used when it is known that the second function will not start before the first. It has the benefit of not running lots of scripts at the same time (using processor power) but only starting them when they are needed.

.lua Script

```
function MoveCar()
        Cmd(0, 100, 916, 1516);
        RunScript("MoveTruck", 4000);
        Suicide();
end;

function MoveTruck()
        Cmd(0, 101, 1583, 223);
        QCmd(0, 101, 1890, 1570);
        QCmd(0, 101, 1100, 1890);
        Suicide();
end;

function Init()
        RunScript("MoveCar", 3000);
end;
```

Explanation

```
function MoveCar()
        Cmd(0, 100, 916, 1516);
        RunScript("MoveTruck", 4000);
```
the second function is now being called here. To make the truck move 7 seconds after the game starts we need to change the time parameter to 4000 because >function MoveCar< already starts 3 seconds into the game
```
        Suicide();
end;

function MoveTruck()
        Cmd(0, 101, 1583, 223);
```

```
        QCmd(0, 101, 1890, 1570);
        QCmd(0, 101, 1100, 1890);
        Suicide();
end;

function Init()
        RunScript("MoveCar", 3000);
```
the run command for >MoveTruck< can now be removed from the Init function
```
end;
```

Lesson 3 – Testing a Condition

This lesson will test to see if a condition has been met and then trigger a new action in response. This is a very useful and commonly used device in Blitzkrieg maps.

Lesson3 Map

This is the Lesson2 map with a ScriptArea called >Farm< marked around the building and two squads of reinforcements with the ScriptIDs of >103< and >104<. These belong to **reinforcement group** >103<. In MapEditor click on the Reinforcement Groups tab to see that the group >103< contains two sets of ScriptIDs. This is necessary as we will order the two squads to do different things, and so the will need different IDs. The building now has a ScriptID >10<

Lesson3.lua Script

```
function MoveCar()
	Cmd(0, 100, 916, 1516);
	RunScript("MoveTruck", 4000);
	Suicide();
end;

function MoveTruck()
	Cmd(0, 101, 1583, 223);
	QCmd(0, 101, 1890, 1570);
	QCmd(0, 101, 1100, 1890);
	Suicide();
end;

function FarmCheck()
	if GetNUnitsInArea(0, "Farm") == 2 then
	RunScript("SquadArrive", 1000);
	Suicide();
end;
end;

function SquadArrive()
	LandReinforcement(103);
	RunScript("SquadMove", 2000);
	Suicide();
end;
```

```
function SquadMove()
        Cmd(0, 103, 1530, 1320);
        Cmd(6, 104, 10);
        Suicide();
end;

function Init()
        RunScript("MoveCar", 3000);
        RunScript("FarmCheck", 2000);
end;
```

Explanation

Only the new script elements will be shown from now on.

```
function FarmCheck()
        if GetNUnitsInArea(0, "Farm") == 2 then
```

>if < is the keyword which tests something against a condition and returns one of two possible answers: True or False). In this case the thing tested is the number of units in an area: >GetNUnitInArea< which is qualified by saying which units (>0<, that is the player's units) and by specifying which area(>"Farm"<, do not forget the quote marks). The condition is >==2< which is to say does the number of units in the area equal 2? (see the note at the end of this lesson regarding symbols used). >then< is the keyword which defines what happens if the answer to the test is True. As >then< operates on the next line of text there is no semicolon.

```
        RunScript("SquadArrive", 1000);
```

the consequence of the test being True is that another function (>SquadArrive<) is run. If the answer is False then nothing happens but the script keeps running

```
        Suicide();
end;
```

this end finishes the >if<, as a rule you need one >end< for every >if< plus one for the function

```
end;
```

this end finishes the function

```
function SquadArrive()
        LandReinforcement(103);
```

this calls the reinforcement *group* >103< onto the map

```
        RunScript("SquadMove", 2000);
        Suicide();
end;
```

function SquadMove()
Cmd(0, 103, 1530, 1320);
this is a command for one unit (103) to move (0), to a set of coordinates (1530, 1320)
Cmd(6, 104, 10);
this is a command for the other unit (104) to enter (6) a building (10)
Suicide();
end;

function Init()
RunScript("MoveCar", 3000);
RunScript("FarmCheck", 2000);
this runs the function >FarmCheck< two seconds after the game starts. Because that function is conditional, nothing will happen *until* the condition is met
end;

Tip: if you use the Blitzkrieg SciTE editor it will place a dash mark to the left of any line which requires an >end<. I place all the ends to the left to make it easier to check that there is the same number of ends below as dashes above!

Note
In conditional functions the following symbols are recognized:

a == b	a equals b
a > b	a is greater than b
a < b	a is less than b
a >= b	a is greater than or equal to b
a <= b	a is less than or equal to b
a ~= b	a does not equal b

Note – A Helpful Chunk
The following script chunk can be found on all my scripts because it helps with testing:

```
function DebugView
        Password("Panzerklein");
        DisplayTrace("any text you like");
        ShowActiveScripts();
        -- God(0, 2);
        Suicide();
end;

function Init()
        Runscript("DebugView", 1000);
end;
```

Explanation

function DebugView
Password("Panzerklein");
this allows other testing functions to work.

> NOTE: if you are running Blitzkrieg with the Stalingrad ailogic.dll file (that corrects units having the ability to fire through buildings) then you must use the following command:
>
> **Password("www.dtf.ru")**

DisplayTrace("any text you like");
shows an in-game message of whatever is inside the quote marks. It is used here because this function will run from the start of the mission so therefore if the message does not appear you will know the script is not running

ShowActiveScripts();
in console mode during the game you can see which script is running. To activate console mode while Blitzkrieg is running press the Tab key

-- God(0, 2);
this is a cheat code. The parameters >0< applies it to the player and >2< specifies that their forces both can kill with one hit and cannot be killed themselves.
This requires Password to work. The two dashes mean that is currently a note and will be

ignored by the script. To make it operate (for testing) just remove the two dashes and save the file.

```
Suicide();
end;

function Init()
        Runscript("DebugView", 1000);
end;
```

DebugView must be started from the Init function just like any other function

Note: when using DisplayTrace the message to be shown must be inside double quote marks ("). Therefore you cannot include double quote marks in the message. You can, however, use single quote marks (').

Lesson 4 – Using Objectives

This lesson covers setting objectives and using **Variables**.

Objectives are the triggered to appear by the script but the text for the objective and the placing of the objective arrows on the minimap is governed by the Resource Editor. How to write and save objectives is covered elsewhere, this lesson will just look at triggering Objective and Objective Complete messages.

Variables are data which is saved by the game for use later. In this lesson we will use Global Variables, information on other sorts is contained in Calvin's Guide.

Lesson4 Map

This is based on Lesson3 map with some AI enemy troops (Player 1) in a trench, and a new ScriptArea called >Trench<.

Lesson4 .lua Script

In this script I have indicated the additions by putting in a note (--NEW--) against whole functions or individual lines.

```
function DebugView()  -- NEW --
        Password("Panzerklein");
        DisplayTrace("Amazing! This really works");
        ShowActiveScripts();
        --God(0, 2);
        Suicide();
end;

function MoveCar()
        Cmd(0, 100, 916, 1516);
        RunScript("MoveTruck", 4000);
        Suicide();
end;

function MoveTruck()
        Cmd(0, 101, 1583, 223);
        QCmd(0, 101, 1890, 1570);
        QCmd(0, 101, 1100, 1890);
        RunScript("ReavealObjective0", 10000);  -- NEW --
        Suicide();
end;

function ReavealObjective0()  -- NEW --
```

```
        ObjectiveChanged(0, 0);
        DisplayTrace("Off we go");
        RunScript("Objective0Check", 2000);  -- NEW -- RunScript("Objective0Complete",
        3000);  -- NEW --
        Suicide()
end;

function FarmCheck()
        if GetNUnitsInArea(0, "Farm") == 2 then
        RunScript("SquadArrive", 1000);
        Suicide();
end;
end;

function SquadArrive()
        LandReinforcement(103);
        RunScript("SquadMove", 2000);
        Suicide();
end;

function SquadMove()
        Cmd(3, 103, 2530, 2850);
        Cmd(6, 104, 10);
        Suicide();
end;

function Objective0Check()  -- NEW --
        if GetNScriptUnitsInArea(130, "Trench") >=1 then
        SetIGlobalVar("Mission0", 1);
        Suicide();
end;
end;

function Objective0Complete()  -- NEW --
        if GetIGlobalVar("Mission0", 0) == 1 then
        ObjectiveChanged(0, 1);
        Suicide()
end;
end;

function Init()
        RunScript("DebugView", 1000);
        RunScript("MoveCar", 3000);
        RunScript("FarmCheck", 2000);
end;
```

Explanation

function ReavealObjective0() -- NEW --
ObjectiveChanged(0, 0);
the parameters specify which objective is changed (0), and what the change is, in this case reveal (0)
DisplayTrace("Off we go");
just a message, this could be omitted
RunScript("Objective0Check", 2000); -- NEW –
runs the test which must be met for the objective to complete
RunScript("Objective0Complete", 3000); -- NEW –
starts the function which will display the Objective Complete message
Suicide()
end;

function Objective0Check() -- NEW --
if GetNScriptUnitsInArea(130, "Trench") >=1 then
different from GetNUnitsInArea as it tests whether any members of a specific unit (ID >130<) are in a given ScriptArea (Trench)
SetIGlobalVar("Mission0", 1);
when the condition is met this keyword creates and saves a variable with a name and value as specified in the parameters, in this case the name >"Mission0"< and value >1<. The value can be any whole number.
Suicide();
end;
end;
two ends because this function contains an >if<

function Objective0Complete() -- NEW --
if GetIGlobalVar("Mission0", 0) == 1 then
this keyword recalls the parameters previously saved by >SetIGlobalVar<. The first parameter is the name of the save global variable and the second parameter *is always zero (0)*. So in total this line calls up the value of the variable we have called >"Mission0"< and tests whether that value equals >1<. If the answer is True then the next line of the script will execute. If the answer is False nothing happens except that the script keeps running
ObjectiveChanged(0, 1);
The parameters >(0, 1)< mean that objective number >0< is complete (1)
Suicide()
end;
end;
two ends because this function contains an >if<

Note
It is not necessary to use Global Variables to trigger Objective Complete messages but they are often used later in the script to check that all the objectives have been completed (as we shall see).

The alternative way to complete the objective would just require one function like this:

```
function Objective0Check()  -- NEW --
        if GetNScriptUnitsInArea(130, "Trench") >=1 then
        ObjectiveChanged(0, 1);
        Suicide();
end;
end;
```

Lesson 5 – Putting It All Together

This lesson puts the previous lessons into action with conditions for objectives, different sorts of test and multiple reinforcements. It demonstrates the use of **if… else** and **if… elseif**. It adds Win and Loose conditions and recalls Global Variables to test whether conditions have been met. It could be the basis for many missions with the different tests being used repeatedly with different ScriptAreas and different ScriptIDs.

Lesson5 Map

This is based on Lesson4 map

Lesson5 .lua Script

```
function DebugView()
	Password("Panzerklein");
	DisplayTrace("Lua really isn't scary any more!");
	ShowActiveScripts();
	--God(0, 2);
	Suicide();
end;

function MoveCar()
	Cmd(0, 100, 916, 1516);
	RunScript("MoveTruck", 4000);
	Suicide();
end;

function MoveTruck()
	Cmd(0, 101, 1583, 223);
	QCmd(0, 101, 1890, 1570);
	QCmd(0, 101, 1100, 1890);
	RunScript("ReavealObjective0", 10000);
	Suicide();
end;

function ReavealObjective0()
	ObjectiveChanged(0, 0);
	DisplayTrace("Off we go");
	RunScript("Objective0Check", 2000);
	RunScript("Objective0Complete", 3000);
	Suicide()
end;

function FarmCheck()
	if GetNUnitsInArea(0, "Farm") == 2 then
```

```
        RunScript("SquadArrive", 1000);
        Suicide();
end;
end;

function SquadArrive()
        LandReinforcement(103);
        RunScript("SquadMove", 2000);
        Suicide();
end;

function SquadMove()
        Cmd(3, 103, 2530, 2850);
        Cmd(6, 104, 10);
        Suicide();
end;

function Objective0Check()
        if GetNScriptUnitsInArea(103, "Trench") >= 1 then
        SetIGlobalVar("Mission0", 1);
        Suicide();
end;
end;

function Objective0Complete()
        if GetIGlobalVar("Mission0", 0) == 1 then
        ObjectiveChanged(0, 1);
        RunScript("Objective1", 2000);  -- NEW --
        Suicide();
end;
end;

function Objective1()  -- NEW --
        ObjectiveChanged(1, 0);
        RunScript("Objective1Complete", 3000);
        DisplayTrace("Clear out all the Americans from this area");
        LandReinforcement(105);
        Cmd(3, 105,2335, 3015);
        RunScript("HeavyArmourDeploy", 30000);
        Suicide();
end;
```

```
function Objective1Complete()  -- NEW --
        if GetNUnitsInParty(1) <= 3 then
        ObjectiveChanged(1, 1);
        SetIGlobalVar("Mission1", 1);
        Suicide();
end;
end;

function HeavyArmourDeploy()  -- NEW --
        LandReinforcement(106);
        RunScript("HeavyArmourMove", 2000);
        Suicide();
end;

function HeavyArmourMove()  -- NEW --
        if GetNUnitsInArea(1, "Trench") > 0 then
        Cmd(3, 106, GetScriptAreaParams(“Trench”));
        else Cmd(3, 106, 393, 3777);
        Suicide();
end;
end;

function USCounterattack()  -- NEW --
        if GetNUnitsInArea(1, "Trench") <= 20 then
        RunScript("USTanks", 3000);
        Suicide();
end;
end;

function USTanks()  -- NEW --
        LandReinforcement(200);
        Suicide();
end;

function WinLoose()  -- NEW --
        if GetIGlobalVar("Mission0", 0) * GetIGlobalVar("Mission1", 0) == 1 then
        Win(0);
        elseif GetNUnitsInScriptGroup(103) + GetNUnitsInScriptGroup(105)
+GetNUnitsInScriptGroup(106) == 0 and
        GetIGlobalVar("Mission0", 0) == 1 then
        Loose(0);
        Suicide();
end;
end;
```

```
function Init()
        RunScript("DebugView", 1000);
        RunScript("MoveCar", 3000);
        RunScript("FarmCheck", 2000);
        RunScript("USCounterattack", 2000);  --NEW--
        RunScript("WinLoose", 3000);  --NEW--
end;
```

Explanation

```
function Objective1()  -- NEW --
        ObjectiveChanged(1, 0);
```

as with >Objective0< explained in Lesson4, this triggers the objective message and arrow for objective 1

```
        RunScript("Objective1Complete", 3000);
```

start the function (in 3 seconds) which will test whether objective is completed

```
        DisplayTrace("Clear out all the Americans from this area");
```

just a message, could be omitted as the objective message will be displayed in finished game

```
        LandReinforcement(105);
```

triggers the arrival of light AFVs (ScriptGroup>105<)

```
        Cmd(3, 105, 2335, 3015);
```

moves the reinforcements (>105<) to new x,y coordinates

```
        RunScript("HeavyArmourDeploy", 30000);
```

starts the function >HeavyArmourDeploy< after 30 seconds (>30000< milliseconds)

```
        Suicide();
end;
```

function Objective1Complete() -- NEW --
if GetNUnitsInParty(1) <= 3 then
tests whether the number of units in the AI enemy's side (>1<) is less than or equal to 3
ObjectiveChanged(1, 1);
if the answer to the test is True then objective >1< is complete (>1<)
SetIGlobalVar("Mission1", 1);
when the condition is met this keyword creates and saves a variable with a name and value as specified in the parameters, in this case the name >"Mission1"< and value >1<
Suicide();
end;
end;
two ends because this function contains an >if<

function HeavyArmourDeploy() -- NEW --
LandReinforcement(106);
triggers the arrival of tanks (ScriptGroup>106<)
RunScript("HeavyArmourMove", 2000);
starts the function (in 2 seconds) which will choose where the tanks go
Suicide();
end;

function HeavyArmourMove() -- NEW --
if GetNUnitsInArea(1, "Trench") > 0 then
tests whether the number of units in the AI enemy's side (>1<) in the area >Trench< is greater than 0
Cmd(3, 106, GetScriptAreaParams("Trench"));
if the answer to the test is True then the tanks (>106<) move aggressively (>3<) to the coordinates of the ScriptArea >Trench<
else Cmd(3, 106, 393, 3777);
>else< gives the instruction if the answer to the test is False (often this isn't given and the result of a False test is *do nothing*), in this case the tanks (>106<) move aggressively (>3<) to new x,y coordinates (>393, 3777<)
Suicide();
end;
end;
two ends because this function contains an >if<

function USCounterattack() -- NEW --
if GetNUnitsInArea(1, "Trench") <= 20 then
RunScript("USTanks", 3000);
Suicide();
end;
end;
two ends because this function contains an >if<

function USTanks() -- NEW --
LandReinforcement(200);
Suicide();
end;

function WinLoose() -- NEW –
if GetIGlobalVar("Mission0", 0) * GetIGlobalVar("Mission1", 0) == 1 then
recalls the Global Variables >Mission0< and >Mission1< (which are set to 1 when the each of the objectives are completed) and tests whether they are both >1< (one multiplied by one equals one)
Win(0);
if the answer to this first test is True then the player (>0<) wins and the You Won message is displayed
elseif GetNUnitsInScriptGroup(103) + GetNUnitsInScriptGroup(105) +GetNUnitsInScriptGroup(106) == 0 and
if the answer to the first test is False then >elseif< performs a second test, in this case counting whether certain of the player's units have been killed. The >and< links this test with another and both must be True for the consequence part of the script to run
GetIGlobalVar("Mission0", 0) == 1 then
tests whether the first objective is complete. This is needed because >function WinLoose()< runs throughout the game (started by >function Init<) and the player's reinforcement do not arrive until after the first objective is complete. Therefore, if we didn't have this additional test then this function would run 3 seconds after the game begins, find that the number of player reinforcements equals 0 and then run the next line
Loose(0);
if the answer to the second test is True then the player looses and the You Lost message is displayed
Suicide();
end;
end;
two ends because this function contains an >if<, but only two because >elseif< does not count as another logical function, it is only the outcome of a False answer to the >if< test

function Init()
RunScript("DebugView", 1000);
RunScript("MoveCar", 3000);
RunScript("FarmCheck", 2000);
RunScript("USCounterattack", 2000); --NEW--
RunScript("WinLoose", 3000); --NEW--
end;

Note: the WinLoose function could also be written as follows:

```
function WinLoose()  -- NEW --
        if GetIGlobalVar("Mission0", 0) * GetIGlobalVar("Mission1", 0) == 1 then
        Win(0);
        if GetNUnitsInScriptGroup(103) + GetNUnitsInScriptGroup(105)
+GetNUnitsInScriptGroup(106) == 0 and
        GetIGlobalVar("Mission0", 0) == 1 then
        Loose(0);
        Suicide();
end;
end;
end;
```

three ends: one for each >if< and one for the function

LessonPlanes

This lesson will demonstrate how to use the script to call aircraft in response to a map event.

LessonPlanes Map

There is a defensive position surrounded by a MapArea called >Bunker<. Aviation has been allocated to the player side in Map Editor using the Edit Units Creation Information button (it looks like… well I'm not sure… a sort of upside-down mushroom).

Click the button to call up the menu, then click on the >+< next to >Player[0]< then click the >+< next to >Aviation<. You can then select which aircraft (and how many) will make up each type of aviation call. The **most important** thing to select is >Appear points<: you must set this to be able to use aviation. Double click the line and then select >Add<. You can now enter the x,y coordinates for the player's aircraft to appear. Notice that these are given as VIS tiles, not Script points. The VIS tile coordinates are displayed in the centre at the bottom of the screen in MapEditor, right next to where the Script point coordinates ate given. Once you have set the appear point, close the menus.

LessonPlanes.lua Script

```
function DebugView()
	Password("Panzerklein");
	DisplayTrace("Here come the planes");
	ShowActiveScripts();
	--God(0, 2);
	Suicide();
end;

function PlaneTrigger()
	if GetNUnitsInArea(1, "Bunker") >= 1 then
	Cmd(36, 999, 0, 1100, 2070);
--	Cmd(36, 999, 0, GetScriptAreaParams("Bunker"));
	Suicide();
end;
end;

function Init()
	RunScript("DebugView", 1000);
	RunScript("PlaneTrigger", 3000);
end;
```

Explanation

function PlaneTrigger()

if GetNUnitsInArea(1, "Bunker") >= 1 then

a conditional test which will trigger the plane call when the number of AI enemy units (player 1) in the ScripArea >Bunker< is greater than or equal to 1.

Cmd(36, 999, 0, 1100, 2070);

calls the aircraft. >36< is the code for Ground Attack aircraft (see Calvin for the full list), >999< is a temporary ID for the aircraft (it could be any number but by convention mapmakers tend to reserve 99, 999 or 9999 for aircraft), >0< is the player's side, >1100, 2070< is the x,y coordinate for the plane to go to (given in Script points)

-- Cmd(36, 999, 0, GetScriptAreaParams("Bunker"));

this line is not being used because it has two dashes at the beginning. It is an alternative way to call the aircraft. Instead of giving the x,y coordinates the script will get the coordinates of the ScriptArea >Bunker< automatically

Suicide();

end;

end;

two ends: one for the >if< and one for the function

LessonPatrol

This lesson will demonstrate how to create a patrol route. In this example only one unit is in the patrol, but it could include any number. If the patrolling unit(s) has an aggressive capacity, then it will engage any enemy units it encounters and then continue on its route (if it survives).

LessonPlanes Map

There is an armored car with the ScriptID >100< in a ScriptArea called >Starting Point<.

LessonPlanes.lua Script

Note: This script was originally provided by Wespex in answer to a question on **the Blitzkrieg Portal forums**, together with an alternative means of achieving the same end.

```
function PatrolArea()
        if GetNScriptUnitsInArea(100, "Starting Point") > 0 then
        Cmd (3, 100, 765, 681);
        QCmd (3, 100, 288, 356);
        QCmd (3, 100, 545, 1145);
        QCmd (3, 100, 299, 1807);
        QCmd (3, 100, 1631, 1479);
        QCmd (3, 100, 1351, 647);
        RunScript("PatrolArea", 2000);
        Suicide();
end;
end;

function Init()
        RunScript("PatrolArea", 5000);
end;
```

Explanation

function PatrolArea()

if GetNScriptUnitsInArea(100, "Starting Point") > 0 then

checks that the patrolling unit(ScriptID >100<) still exists and is at the initial position (ScriptArea >Starting Point<)

Cmd (3, 100, 765, 681);

orders aggressive move (>3<) by unit >100< to new x,y coordinates (>765, 681<)

QCmd (3, 100, 288, 356);

orders aggressive move (>3<) by unit >100< to new x,y coordinates once previous order is complete (QCmd)

QCmd (3, 100, 545, 1145);

as above

QCmd (3, 100, 299, 1807);

as above

QCmd (3, 100, 1631, 1479);

as above

QCmd (3, 100, 1351, 647);

as above but as this is the last movement order the final coordinates must be inside the ScriptArea >Starting Point<

RunScript("PatrolArea", 2000);

repeats the whole script

Suicide();

end;

end;

two ends: one for the >if< and one for the function

Note: A patrol can have any number of movements in each circuit simply by adding or removing >QCmd(3, ScriptID, x, y)< lines. The important thing is that the final movement must bring the patrolling unit back into the ScriptArea which is tested in the first line of the function.

Debugging Scripts

Debugging is the process of finding and eliminating bugs in code. In the world of programming, the recognized steps involved in debugging are:

Recognize that a bug exists
Isolate the source of the bug
Identify the cause of the bug
Determine a fix for the bug
Apply the fix and test it

Scripts will need to be debugged if they are either “broken” – that is, they do not run at all, or if they do not behave as expected.

Broken Scripts

You will be aware that your script has been "broken" by one or more errors when any objectives triggered by the script are not displayed in the game. It is good practise to put in a **DisplayTrace** statement in the **Init** function to display some text such as “Script working…”. If the script is OK this will appear straightaway. If it does not appear then you know the script is faulty. The statement can be removed once the script is debugged or commented out.

TIP #1: when you find that your script is broken you do not have to close Blitzkrieg. You can return to Windows with the game still running, debug and edit the script, save it and then choose the "Restart Mission" option in Blitzkrieg to test it again.
NOTE: This will not work if your script file is inside a .pak file as these files are locked when the editor and game are active.

TIP #2: Do not compile the map into a .pak file until it is fully complete and debugged. Any map open in the editor that is stored inside a .pak file is saved directly to the Data folder from the editor and could therefore easily end up being excluded from your .pak file unless you remember to keep the contents up-to-date.

When debugging your script you should check for the following, most common, mistakes:

Functions not terminated with an "**end”;** statement

Comment lines incorrectly started using a single dash – “-“ – instead of a double-dash – “--“

Condition checking blocks not being terminated with their own "**end;"** statements.

Example

```
function CheckTown()
        if GetNUnitsInArea(0, "Town") == 0 then
        ObjectiveChanged(0,1);
        RunScript( "Objective1", 5000);
end;
```

The above code would break a script because the "**if GetNUnits...**" block is missing its own "**end;**" statement.
Every **function** block requires its own **end;** statement. Every **if** block requires its own **end;** as well.

Incorrect usage of case: the **if** command must be typed in lowercase , it should not be entered as **If**

Statements not terminated with a semi-colon – ";".
Note: this does not apply to **function** and **if** keywords.

Text values (or strings) that have not been terminated correctly. The following, for example, would break a script - **GetUnitsInArea(0, "Town) –** because the script area name **Town** was not closed with quotation marks.

Use of quotes in "**DisplayTrace**" statements.
Example: this code is invalid: **DisplayTrace("Commander, the "Tigers" have arrived.");** because wrapping the word **Tigers** in quotation marks has indicated to the script processor that the string you want to display is **"Commander, the "**

Functions and checks embedded in brackets - "(" and ")" – that do not contain the correct number of opening or closing brackets.
This example below is incorrect:

```
if (GetNScriptUnitsInArea (1, "finalattack") == 1 then
```

because the **GetNScriptUnitsInArea** function has not been terminated with a closed bracket symbol.

Typing **Lose();** instead of **Loose();**. It might be tempting to write this using the correct spelling but **Loose** is the correct name.

Win() statements missing the player number.

To save time during the debugging process it is useful to know that the following do not break scripts:

Landing a non-existent reinforcement group
Calling a non-existent function (using the **RunScript** command)

Debugging Scripts That Behave Unexpectedly

Sometimes you will find that your script does not behave exactly as expected. For example, objectives might not be completed when you have fulfilled the requirements or reinforcement groups may appear repeatedly. Below are some common mistakes to look for in your script:

If you find that a particular action or event (for example, a reinforcement group keeps appearing on the map) is repeating itself then you should check for missing **Suicide();** statements. This statement tells a script to stop processing and its use is critical for ensuring that scripts execute smoothly.

Example

```
function Init()
        RunScript("MyCheck", 5000);
        …
end;

function MyCheck()
        if GetNUnitsInArea(0, "Town") > 0 then
                RunScript(Reinforcements,5000);
        end;
end;

function Reinforcements()
        LandReinforcement(100)
end;
```

The above code runs the **MyCheck** function every five seconds. This function checks whether the player (side 0) has one or more units in the map area defined as "Town". If he does then the **Reinforcements** function is called with the intention of calling in reinforcement group 100. However, in the above code, reinforcement group 100 would appear repeatedly whenever there are player 0 units in the "Town" area because the **MyCheck** function was not told to stop running after the condition was fulfilled.

In this example, the correct code for **MyCheck** should read:

```
function MyCheck()
if GetNUnitsInArea(0, "Town") > 0 then
        RunScript(Reinforcements,5000);
        Suicide();
end;
end;
```

The **Suicide();** statement tells the script processor to stop running the **MyCheck** script once the conditions have been met and the **Reinforcements** script has been called.

"Objective Received: Unknown Objective"
This will appear on your map if the relevant objective entry is missing from your map's 1.xml file. However, this message is perfectly normal if running your map directly from the map editor as, in this circumstance, Blitzkrieg does not process the files in the **scenarios\custom\missions*mission name*** folder.

If you are checking for units in a particular area and the checks are failing to return the desired results then double-check that you do not have two or more map areas defined with the same name.

If aviation units do not appear after being called with one of the **Cmd** functions (e.g. **example here**) then check that the aviation was not previously disabled elsewhere in the script via the **DisableAviation()** function.

Appendix A – Blitzkrieg Function List (by Wespex)

Custom Function	Input Value	Return Value
AddIronMan	(iScriptID)	-
ChangeFormation	(iScriptID, iFormation)	-
ChangePlayer	(iScriptID, iParty)	-
ChangeWarFog	(iParty)	-
Cmd|GiveCommand	(iAction, iScriptID, xLoc, yLoc)	-
DamageObject	(iScriptID, fDamage)	-
DeleteReinforcement	(iScriptID)	-
DisableAviation	(iParty, iAviationType)	-
DisplayTrace	(strText, params, ...)	Text on screen
Draw	-	-
EnableAviation	(iParty, iAviationType)	-
FlagReinforcement	(nParty)	-
GetActiveShellType	(iScriptID)	Shell Type
GetAviationState	(iPlayer)	Last aviation call by player
GetFGlobalVar	(strGlobalVarName, 0)	strGlobalVarName & value
GetFrontDir	(iScriptID)	Direction unit Is facing
GetIGlobalVar	(strGlobalVarName,0)	strGlobalVarName & value
GetMapSize	-	Size of map in script points
GetNAmmo	(iScriptID)	ammunition levels (pri, sec)
GetNAntitankInScriptArea	(strScriptAreaName)	Number of anti-tank obsticals in area
GetNAPFencesInScriptArea	(strScriptAreaName)	Number of barb wire pieces in area
GetNFencesInScriptArea	(strScriptAreaName)	Number of fence pieces in area
GetNMinesInScriptArea	(strScriptAreaName)	Number of mines in area (at & ap)
GetNScriptUnitsInArea	(iScriptID, strScriptAreaName)	Number of scriptid units in area
GetNTrenchesInScriptArea	(strScriptAreaName)	Number of trench pieces in area
GetNUnitsInArea	(iPlayer, strScriptAreaName)	Number of player units in area
GetNUnitsInCircle	(iPlayer, X, Y, Radius)	Number of player units in co-ords
GetNUnitsInParty	(iPlayer)	Number of player units on map
GetNUnitsInScriptGroup	(iScriptID, iPlayer)	Number of scriptid units
GetNUnitsOfType	(strUnitType, iPlayer)	Type of infantry
GetObjCoord	(iScriptID)	Object's location
GetObjectHPs	(iScriptID_Static)	Object's hit points
GetScriptAreaParams	(strScriptAreaName)	Parameters of area
GetSGlobalVar	(strGlobalVarName, 0)	strGlobalVarName & value
GetSquadInfo	(iScriptID)	Squad formation
GetUnitState	(iScriptID)	Unit state
God	(iParty, iMode)	-
IsEntrenched	(iScriptID)	Entrenchment value of unit
IsFollowing	(iScriptID)	Following value of unit
IsStandGround	(iScriptID)	Stand ground value of unit

Custom Function	Input Value	Return Value
IsWarehouseConnected	(iScript_StorageID)	Small depot connected to main
KillScript	(strScriptFunctionName)	-
LandReinforcement	(iReinfID)	-
Loose	-	-
ObjectiveChanged	(iObjNum, iState)	Objective state
Password	(strName)	-
QCmd|GiveQCommand :	(iAction, iScriptID, params)	-
RandomFloat	-	Decimal between 0 and 1
RandomInt	(n)	Random number from 0 to n
ReserveAviationForTimes	(iParty, iTime)	-
RunScript	(strFunctionName, iTime, iTurns)	-
SetCheatDifficultyLevel	(n)	-
SetDifficultyLevel	(n)	-
SetFGlobalVar	(strGlobalVarName, fVar)	-
SetGameSpeed	(n)	-
SetIGlobalVar	(strGlobalVarName, iVar)	-
SetSGlobalVar	(strGlobalVarName, sVar)	-
ShowActiveScripts	-	Current scripts show in console
Suicide	-	-
SwitchWeather	(iState)	-
SwitchWeatherAutomatic	(iState)	
Trace	(strText, params)	Text in console
ViewZone	(strScriptAreaName, iParam)	-
Win	(iParty)	-

Appendix B – Useful Links

Blitzkrieg SciTE Editor

Calvin's Guide to Lua Programming

Single Player Map Tutorial by Wespex

Other Tutorials at Blitzkrieg Portal

Map Making Questions and Answers at Blitzkrieg Portal Forums