SQF-Handbuch

16.09.2013

Autoren: [3.JgKp]James, [3.JgKp]Bunkerfaust

Eine praxisorientierte Einführung in SQF für ArmA 2 und 3.

Stand: Alpha v04 – Teil I fertig, Teil II und III in Arbeit

Inhalt

EINLEITUNG

Nun also ein Handbuch zu SQF – für die Programmiersprache von ArmA2 und ArmA3. Wie kommt man dazu?

Zum einen ist es mir ein persönliches Anliegen, all jenen, die ebenfalls über die unglaublich vielfältigen Möglichkeiten der Anpassung und Veränderung von ArmA2 staunen, eine Hilfe an die Hand zu geben, diese Möglichkeiten möglichst problemlos und mit wenig Frust kennenzulernen. ArmA ist extrem vielschichtig, und die Anpassungsmöglichkeiten erstrecken sich von einfachen eigenen Missionen ohne jedes Skript bis hin zu ganzen Benutzerdialogen und eigenen Modifikationen oder 3D-Objekten. Diese große Anpassungsfähigkeit basiert auf einer entsprechend großen Offenheit und Adaptierbarkeit des Spiels unter anderem anhand von Skripten. Der einfachste Einstieg, um ArmA2 – auch und gerade im Multiplayer – seinen eigenen Bedürfnissen anzupassen, bietet die spieleigene Skriptsprache SQF. Daher möchte dieses Handbuch dem geneigten Leser als Einstieg dienen, der ohne Vorkenntnisse – oder mit geringen – von Grund auf den Umgang mit SQF erlenen möchte. Ziel ist dabei stets das Schreiben eigener Skripte, das Erlangen von Fähigkeiten, eigene Ideen umzusetzen. Da ich persönlich die Erfahrung gemacht habe, dass man am Anfang extrem auf sich allein gestellt ist und es gefühlte 100 Internetseiten zu diesem Thema, aber keinen klaren Einstiegspunkt gibt, möchte ich gerade der deutschen Community mit diesem Dokument quasi einen Einstieg präsentieren.

Dieses Handbuch ist in drei große, aufeinander aufbauende Teile gegliedert.

Teil I – SQF von A bis Z beschäftigt sich mit der grundlegenden Syntax und dem Aufbau von SQF. Wie ist ein Skript aufgebaut? Welche Befehle gibt es? Welche grundlegenden Programmierfähigkeiten brauche ich? Wie ist ein Skript aufgebaut? Damit liefert Teil I die Grundlage für jeden erfolgreichen Skripteschreiber.

Teil II – SQF im Multiplayer beschäftigt sich mit dem schwierigsten Teil von SQF für viele Anfänger und auch Profis: Der Lokalität. Wann wird welches Skript bei welchen Spieler/Client ausgeführt? Wie kann ich ein Skript für alle auslösen, wie nur für einen bestimmten Clienten? Welche Rolle spielt der Server? Wie sind die Belastungen verteilt?

Teil III – SQF in der Praxis beschäftigt sich zum Schluss mit Praxisbeispielen und Tipps & Tricks für den SQF-Skripter. Hier werden Beispiele präsentiert und größere Codefragmente präsentiert, um gängige oder auch nicht so gängige Lösungen für auftretende Probleme zu präsentieren.

Die Teile bauen inhaltlich natürlich aufeinander auf und sind als Arbeitspakete mit jeweils einem thematischen Schwerpunkt zu verstehen. Wir werden uns also in den Teilen II und III nicht mehr auf die Grundlagen konzentrieren, sondern Teil I weitgehend voraussetzen, also nur noch neue Befehle ausführlicher vorstellen.

Jetzt wird sich aber der ein oder andere mit etwas mehr Vorwissen fragen, wie er den maximalen Nutzen aus diesem Handbuch ziehen kann. Dazu haben wir zwei Anregungen: Zum einen verfügt dieses Handbuch über mehrere Verzeichnisse ganz am Ende, die mit viel Arbeit zusammengestellt wurden. Hier findet ihr alle behandelten Befehle, Datentypen sowie Stichworte, die wir in diesem Handbuch behandeln. Dies kann also ein geeigneter Einstiegspunkt für den Fortgeschrittenen darstellen.

Zum anderen stellen wir euch mit **Kapitel [2]** ein Beispielprojekt vor, dass ihr aktiv nachvollziehen und nachprogrammieren sollt. Hier werden so gut wie alle bisher in Teil I behandelten Konzepte integriert. Wer also dort einsteigen möchte und keine Verständnisprobleme hat, kann direkt zu den Teilen II oder III übergehen.

Verwendete Formate

Wir haben uns um eine größtmögliche Lesbarkeit des Dokumentes bemüht. Aufgrund der Thematik ist es natürlich unablässig, Code-Beispiele und –fragmente zu präsentieren. Dies geschieht in der folgenden abgesetzten Weise:

```
ein Beispielcode;
```

Wo der Code sich über mehrere Zeilen erstreckt, haben wir für euch den Code in Notepad++ vorbereitet und präsentieren euch die Beispiele als Screenshots. Jedes Bild ist dabei immer ein Link zur eigentlichen SQF-Datei, so dass ihr längere Beispiele ohne Probleme mitverfolgen und gleich anpassen/verändern könnt:

Listing_Beispiel

```
1   if (side player == WEST) then { ... }; // Eine einfache if-Abfrage
```

Im Text selbst können ebenfalls z.B. zur Erklärung Code-Fragmente vorkommen. Damit diese nicht mit normalen Begriffen verwechselt werden, wird hier dieselbe Schriftart gewählt: `Code-Fragment`.

Werden im Text Befehl von SQF oder Datentypen genannt, so sind diese in Kapitälchen gesetzt: BEFEHL.

Dieses Dokument lebt zu einem großen Teil von aktiven Verweisen auf das BI Wiki sowie Internetseiten der Community, die immer dem tieferen Verständnis und der Aneignung von Hintergrundwissen dienen. Solche Links sind durch ein Link kenntlich gemacht.

Zur besseren Leserlichkeit wurde aber darauf verzichtet, jedes Wort, das angeklickt werden kann, als Link zu kennzeichnen. Daher sollte man mit der Maus etwas „kreativ sein“ Es gibt generell zwei Faustregeln: Überschriften, die konkrete Befehle oder Datentypen betreffen, sind verlinkt. Bei Überschriften, die als Link anklickbar sind, weißt zudem das Symbol darauf hin.

Neue Befehle, die zum ersten Mal eingeführt werden und deshalb in **orange** erscheinen, sind verlinkt. Auf diese Links könnt ihr einfach Klicken und landet dann direkt beim jeweiligen Wiki-Eintrag. Dies sollte euch bei der Lektüre helfen. Zudem ist jeder Befehl, der so markiert ist, im Anhang im Befehlsverzeichnis aufgeführt. Mitunter stehen auch andere, wichtige Schlüsselbegriffe in **orange**, dann ohne Link.

Die verwendeten Abbildungen stammen bis auf die Titelseiten der Hauptteile sowie den Einträgen aus dem BI Wiki aus eigener Quelle.

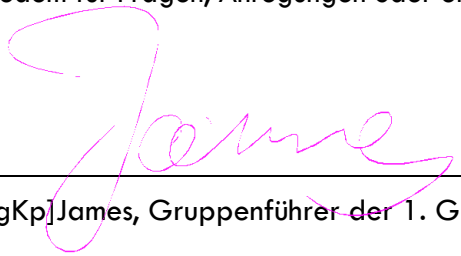
Aufruf zur Mitarbeit

Wir haben dieses Dokument in erster Linie für euch geschrieben. Wir verfolgen damit kein kommerzielles oder persönliches Interesse außer dem Wunsch, allen angehenden Skriptern eine wertvolle Hilfe an die Hand zu geben. Genau aus diesem Grunde bitten wir euch aber um eure aktive Hilfe: Sollte dieses Dokument an einer Stelle sein Ziel verfehlen und durch Fehler, umständliche Erklärungen oder gar fehlende Hilfestellung euch nicht den gewünschten Lernerfolg bringen, bitte kontaktiert uns. Dazu stehen wir euch per E-Mail, im Teamspeak sowie auf unserer Homepage per PM zur Verfügung.

Außerdem stehe ich jedem für Fragen, Anregungen oder eigene Skriptwünsche jeder Zeit zur Verfügung.

In diesem Sinne

07.01.2014



[3.JgKp]James, Gruppenführer der 1. Grp | 1. Zug, Ausbilder, Rekrutenbeauftragter

Kontakt: james.3JgKp@t-online.de | ICQ: 136677992 | Profil

DIE CODEBEISPIELE IN DIESEM HANDBUCH

Wie bereits in der Einleitung erwähnt, wurden alle längeren Codebeispiele als Listings extern erstellt und via Screenshot für dieses Handbuch bereitgestellt. Gleichzeitig dient jeder Screenshot als Hyperlink zur SQF-Datei, so dass ihr gleich per Linksklick die Datei als Sourcecode für euch selbst downloaden könnt.

Aus didaktischen Gründen wird Notepad++ mit aktivierter SQF-Syntaxhervorhebung verwendet (siehe Kapitel I. 1Vorbereitungen). Da mir persönlich aber die Hervorhebung farblich zu schwach ausfällt, habe ich selbst eine angepasste XML-Datei für die Sprache SQF geschrieben, die auch einige erweiterte Befehle enthält. Diese könnt ihr euch gerne ebenfalls herunterladen. Eingebunden wird die Datei über `Sprachen >> Eigene Sprache definieren... >> Importieren` (XML-Datei auswählen).

www.workareal.de/3.jgkp/SQF/Skripte/SQF(eigen).xml

Wichtig ist, dass wir kurz die verwendeten Farben gemeinsam definieren, damit die Beispiele klar nachvollziehbar bleiben:

Listing_Definitionen.sqf

```sqf
hint, str          // SQF-Befehle: schwarz
!=, >,<, =, ;      // SQF-Operatoren: schwarz
if, for, while     // Strukturbefehle: blau fett
BIS_fnc_help       // Funktionen: orange
(), {}, []         // Strukturklammern: rot
#define            // Makros: schwarz fett
// hi, /* ... */// Kommentare: grau

array, a, b        // globale Variablen: lila fett
_array, _zahl      // lokale Variablen: lila schwach
1,2,99             // Datentyp Number: blau schwach
"Hallo Welt!"      // Datentyp String: grün fett
west, east         // Datentyp Side: blau schwach
true, false        // Boole'sche Variablen: blau fett
```

Alle Listings dieses Handbuchs

Alle Codebeispiele, die länger als eine Zeile sind, wurden in Notepad++ vorgeschrieben und euch als Hyperlinks bei den Screenshots zur Verfügung gestellt. Hier ein Link zu allen Beispielen in einer zip-Datei:

www.workareal.de/3.jgkp/SQF/Skripte/SQF-Handbuch_Listings.zip

Teil I – SQF von A bis Z

I. 1. VORBEREITUNGEN

A KLEINE HELFERLEIN

Willkommen zu diesem Kurs, willkommen in unserem Handbuch und ganz besonders willkommen unter all jenen, die sich für Scripting und Missionsbau interessieren. Scripting ist für viele eine hohe Kunst und wir wollen keinen Hehl daraus machen, dass wir über durchaus komplizierte und anspruchsvolle Konstrukte und Scripte sprechen werden. Aber wir möchten den Einstieg bewusst einfach halten und damit wirklich jeden Leser und jede Leserin ermuntern, uns in die Welt von SQF zu folgen. Wer auf sich alleine gestellt ist und die ersten Schritte alleine gehen will oder muss, findet einen steinigen und holprigen Weg. Wir möchten diesen Einstieg glätten und so einfach gestalten, dass jeder uns folgen und die Scripte direkt selbst umsetzen kann.

Allerdings sind einige Voraussetzungen für das Scripten notwendig, sofern wir nicht Alles direkt im Editor erledigen wollen.

Zum einfachen Betrachten, Erstellen und Bearbeiten einer SQF-Datei ist nicht viel notwendig – der mitgelieferte Windows Editor reicht dafür aus. Wenn wir uns aber etwas mehr Komfort wünschen – und dazu zählen Dinge wie die Auto-Vervollständigung und die Hervorhebung der Syntax.

Wir möchten daher zunächst zwei Editoren vorstellen, die sich in der Praxis bewährt haben.

Notepad++

Das Programm Notepad++ ist ein sehr mächtiger und dabei doch einfacher und übersichtlicher Editor. Er bietet gegenüber dem Windows Editor sehr viele Komfort-Funktionen wie Zeilennummerierung, mehrere gleichzeitig geöffnete Dateien und Hervorhebung. Das Programm kann unter

http://notepad-plus-plus.org/

heruntergeladen werden. Notepad++ ist so, wie es kommt, bereits völlig ausreichend, um damit Scripte zu verfassen. Wer aber wirklich Notepad++ voll ausschöpfen möchte, braucht noch folgende Erweiterung:

http://www.armaholic.com/page.php?id=8680.

Der Download umfasst nach dem Entpacken zwei wichtige Dateien in den Ordnern AUTOCOMPLETION und SYNTAXHIGHLIGHTING, die im Programm entsprechend der Installationsanweisungen eingepflegt werden müssen. Nach der erfolgreichen Installation stehen die Autovervollständig sowie eine umfangreiche Funktionsbibliothek samt Hervorhebung der Syntax zur Verfügung.

SQF Editor

Eine „fertige“ Alternative bietet der SQF Editor von Amra:

http://www.armaholic.com/page.php?id=14715.

Dieser ähnelt stark der verbreiteten IDE Netbeans für JAVA und bietet all dessen bekannte Funktionen, unter anderem die Organisation in Projekten, Autovervollständigung, farbige Hervorhebung, Funktionsbibliothek usw. Der Editor ist eine standalone Version und startet recht zügig. Einziger Nachteil ist die Installation des JDK (ebenfalls im obigen Link zu finden), der Entwicklungsumgebung von JAVA.

B VERSCHIEDENE SCRIPTREFERENZEN

Da wir jetzt über die nötige Software verfügen, müssen wir uns noch darüber Gedanken machen, wo wir eigentlich die Befehle herbekommen. Bei weit verbreiteten Programmiersprachen übernimmt dies meist ein Wiki oder direkt eine Dokumentation des Herstellers (siehe JavaDoc). Bei SQF müssen wir dazu leider auf mehrere Quellen zurückgreifen. Nichtsdestotrotz leistet das Wiki von *Bohemia Interactive* sehr gute Dienste:

Scripting Commands ArmA2

http://community.bistudio.com/wiki/Category:Scripting_Commands_ArmA2

OFPEC Scripting Command Reference

http://www.ofpec.com/COMREF/

Command Reference

http://www.arma2.com/comref/comref.html

Hier sollte für jeden Geschmack etwas dabei sein. Die letzte Referenz bietet die Möglichkeit, direkt per Sucheingabe nach verschiedenen Befehlen zu suchen oder die eingebauten Menüpunkte oben links als Startpunkt zu benutzen.

Wie wir diese Datenbanken nutzen bzw. bei Befehlen die relevanten Informationen herausholen, soll uns später bei konkreten Beispielen interessieren, hier soll nur die Quelle bekannt gemacht werden!

Wer wie ich ein kleines Programm favorisiert, anstatt im Internet herumzuklicken, kann folgende Alternative ausprobieren:

ArmA2 Command Lookup Tool

http://www.armaholic.com/page.php?id=14016

Dieses kleine Programm ist etwas veraltet, aber die Seite erklärt, wie ihr die neuesten Befehle abrufen könnt: Ihr speichert euch die Scripting Commands-Seite als HTM-Datei und führt das Programm mittels einer Verknüpfung und dem Startparameter –generatelist aus. Daraufhin wird euch das Programm über den Status des Vorgangs informieren. Lief alles reibungslos, könnt ihr die ArmaCommandLookup.exe starten und z.B. den Befehl CALL finden, der vorher nicht enthalten war.

Damit genug der Worte, starten wir!

I. 2. EIN PRAKTISCHER EINSTIEG IN SQF

In der Welt der Programmiersprachen hat es sich eingebürgert, sein erstes Programm dafür zu verwenden, die Botschaft „Hallo Welt“ auszugeben. Dem wollen wir uns nicht verschließen und uns in die Reihe von Millionen Anfängern einreihen. Daher zunächst der Code:

```
hint "Hallo Welt!";
```

Sieht einfach aus? Ist es auch! Bevor wir uns darum kümmern, wie diese Zeilen jetzt im Spiel zum Leben erweckt werden, sollten wir uns kurz der Syntax widmen. Mit Syntax meinen wir:

UNTER SYNTAX VERSTEHT MAN IN DER GRAMMATIK DIE SATZLEHRE.

Syntax bedeutet in unserem Zusammenhang immer die korrekte Aneinanderreihung von Befehlen. SQF erwartet – wie jede Programmiersprache – eine bestimmte Syntax – also eine bestimmte Reihenfolge. Ein Befehl erwartet ein Argument, das übergeben wird, eine Zuweisung erwartet den Wert und die Variable, die den Wert repräsentieren soll. Es gibt Befehle, die erfordern jeweils vor und nach ihrem Auftreten ein Argument, es gibt andere, die erwarten nur nach ihrem Auftreten ein Argument und wenige Befehle stehen auch allein. All dies gilt es zu berücksichtigen, wenn wir einen Befehl benutzen. Daher schauen wir uns jetzt einmal die Informationen zum Befehl **hint** an, die uns z.B. das BI WIKI liefert[1]:

Description

Description: Outputs a hint message on the left upper corner of the screen (in OFP with a ding sound).
The text may contain several lines.

Syntax

Syntax: **hint** text

Parameters: text: String - the message to write on the screen may consist of any characters. \n indicates a line break.

Return Value: Nothing

Examples

Example 1:

```
hint "Press W to move forward. \nPress S to move backwards."
```

outputs the following message:

```
Press W to move forward.
Press S to move backwards.
```

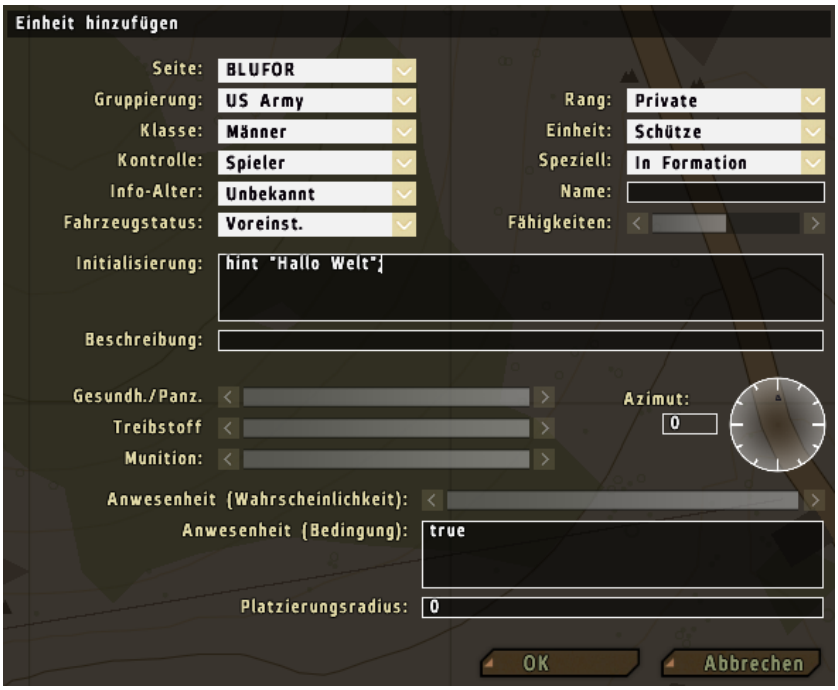
Known Problems: Avoid hint messages that exceed the screen, as this may lead to crashes.

Diese Aufteilung finden wir in den meisten „Nachschalgewerken“ und sie hilft uns, die relevante Information schnell aufzufinden. Zunächst verrät uns die Description, was der Befehl überhaupt tut (auch wenn hier ein Fehler passiert ist, da HINT oben rechts erscheint). Danach können wir uns die Syntax ansehen, die beschreibt, wie der Befehl aufgebaut ist und eingesetzt werden muss. Hier ist es also so, dass zunächst der Befehl kommt und danach der Text. Dieser muss vom Typ „String“ sein, wie uns die nächste Zeile Parameter verrät. Zu den Datentypen kommen wir allerdings etwas später. Wichtig ist an dieser Stelle nur, dass STRING Zeichenketten sind, die in Anführungszeichen (“ “) eingeschlossen werden. Der Rückgabewert wird uns später beschäftigen, hier brauchen wir ihn noch nicht.

[1] Nach dieser Stelle gehen wir davon aus, dass der Leser dies jetzt immer tun wird, auch ohne nochmalige Aufforderung.

Die für den Praktiker oft nützlichste Information ist allerdings danach in den Beispielen und weiterführenden Informationen/Kommentaren zu finden, denn hier stehen oft Anwendungsbeispiele, die auch verwandte Befehle und Konzepte mit einbeziehen. So erfahren wir, dass der Ausdruck \n einen Zeilenumbruch erzeugt. Viel mehr können wir an dieser Stelle von diesem Befehl nicht erwarten – und es reicht für unsere Zwecke.

Widmen wir uns daher jetzt der Ausgabe im Spiel. Hier haben wir zunächst zwei grundlegende Möglichkeiten, ohne auch nur ein externes Script schreiben zu müssen: Die Init-Zeile und Auslöser. Jedes Editor-Objekt, das wir auf die Karte setzen, bietet ein sogenanntes Initialisierungsfeld, das grundsätzlich bei Missionsstart ausgelesen/ausgeführt wird. Tragen wir hier unseren Code (oder z.B. Zeilen für die Ausrüstung, für den Startzustand einer Einheit etc.) ein, so können wir erwarten, dass er sofort ausgeführt wird. Genau dies können wir als Erstes testen.



Diese einfachste Variante liefert bereits das gewünschte Ergebnis, wie man sich via Vorschau überzeugen kann.

Die zweite Möglichkeit besteht im Verwenden von Auslösern. Diese bieten bereits drei Felder, die Scripte enthalten dürfen: Die Bedingung, die Aktivierung und die Deaktivierung. Alle drei Felder sind verkappte Compiler für Scripte und können später quasi als Ersatz für bestimmte Script-Konstrukte wie Bedingungen oder Schleifen benutzt werden. Uns interessiert zunächst nur die einfache Anwendung. Dazu könnten wir einen Auslöser, der bei West-Einheiten auslöst, verwenden:

Auch dieser Auslöser funktioniert – sofern wir selbst West spielen – komplikationsfrei.

Die dritte und später häufigste Möglichkeit, ein Script zu starten, erfolgt jedoch über einen Befehl, der SQF-Dateien im Spiel ausführt. Dieser lautet **execVM**. Wir betrachten zunächst wieder die Wiki-Seite:

Description

Description: Compile and execute SQF Script.

The **optional** argument is passed to the script as local variable _this.

Script is compiled every time you use this command.

The Script is first searched for in the mission folder, then in the campaign scripts folder and finally in the global scripts folder.

Syntax

Syntax: Script = argument **execVM** filename

Parameters: argument: Any Value(s)

filename: String

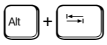
Return Value: Script - script handle, which can be used to determine (via scriptDone) when the called script has finished.

Examples

Example 1:

```
_handle = player execVM "test.sqf";
waitUntil {scriptDone _Handle};
```

Wer sich weiter einlesen möchte, ist herzlich aufgefordert, den Links im Wiki zu folgen, dafür ist es schließlich da! Uns interessiert zunächst wieder die Funktion: Der Befehl kompiliert und führt auch ein SQF-Script aus. Des Weiteren erfahren wir, dass es offenbar optionale Parameter gibt und dann das Script im Missions-Ordner gesucht wird. Wenn wir also ein Script im Spiel einbinden wollen – was, wie gesagt, später unser Hauptweg sein wird, auf dem wir Scripte starten werden – dann müssen wir das Script als solches zunächst im Missionsordner anlegen. Wer nicht weiß, wo dieser ist: Er befindet sich in Eigene Dokumente bzw. `Documents >> ArmA 2 Other Profiles >> Euer Benutzer-Ordner >> missions >> eure Mission`. Dazu solltet ihr sie natürlich zunächst im Editor unter Benutzereigene Missionen abgespeichert haben. Eine jungfräuliche Mission enthält in ihrem Ordner immer nur eine MISSION.SQM. Diese Datei ist extrem wichtig für das Spiel, da sie alle gesetzten Einheiten, Auslöser, Objekte etc. enthält und damit quasi eure Mission in Textform enthält. Wer hier herumpfuscht, wird seine Mission nicht mehr laden oder retten können, also vorerst Finger weg.

Ganz wichtig ist an dieser Stelle die Bemerkung, dass Scripte im Spiel ohne Probleme zur Laufzeit bearbeitet werden können. Das heißt, wenn ihr im Spiel seid und gleichzeitig scripten wollt: Kein Problem! Einfach mit  auf den Desktop wechseln und das Script im Editor bearbeiten. Nach dem Abspeichern kann das Script im Spiel direkt gestartet bzw. neu aufgerufen werden.[2]

Wenn wir uns jetzt also im selben Ordner wie die MISSION.SQM befinden, dann müssen wir lediglich eine neue Textdatei (Rechtsklick >> Neu) anlegen. Diese wird aber zunächst noch eine txt-Datei sein. Solltet ihr die Dateiendung nicht sehen können, müsst ihr über die Ordner-Ansichtsoptionen zunächst Dateiendungen einblenden lassen. Habt ihr die Dateiendung, dann könnt ihr eure Text-Datei einfach in die Endung .sqf abändern. Es bleibt eine Textdatei und wir können sie mit jedem Editor öffnen.

Als nächstes öffnen wir die Datei mit einem Editor unserer Wahl (ich werde in Zukunft Screenshots verwenden, die aus Notepad++ oder dem SQF-Editor stammen). Danach kopieren wir den einzeiligen Code, den wir bisher verwendet haben, hinein und speichern die Datei. Das war's, mehr braucht auch das Script nicht (wie haben ja bereits gesehen, dass der Code funktioniert). Was allerdings fehlt, ist die Einbindung ins Spiel.

Wir müssen nun also im Spiel/Editor irgendwo festlegen, wann und wie unser Script geladen werden soll. Der Befehl dazu lautete (s.o.)

```
Script = argument execVM filename
```

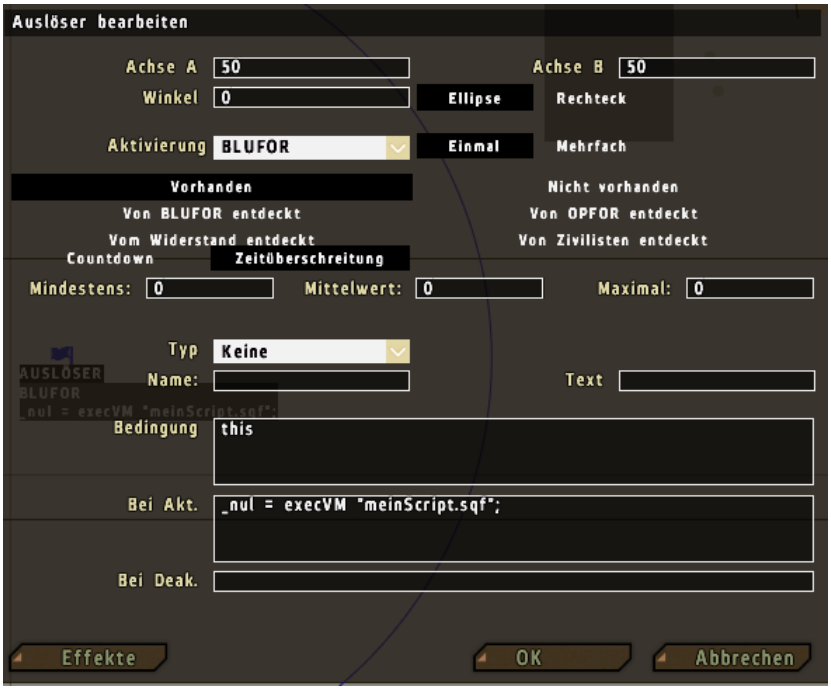
Wir hatten außerdem bereits nachgelesen, dass **argument** optional ist, also nicht notwendigerweise für einen Aufruf gebraucht wird. Um erst später auf die Feinheiten wir Übergabe- und Rückgabeparameter eingehen zu müssen, möchten wir an der Stelle einfach folgenden Aufruf vorgeben:

```
_nul = execVM "meinScript.sqf";
```

Wir werden im nächsten Kapitel besprechen, warum wir manchmal am Ende der Zeile ein ; (Semikolon) haben und manchmal nicht[3]. Jetzt widmen wir uns nur dem Aufruf. In unserem Beispiel heißt unser erstes Script genau so: MEINSCRIPT.SQF. Ihr müsst natürlich euren Dateinamen entsprechend anpassen. Diese Zeile kopieren wir jetzt z.B. wieder in die Aktivierungszeile des Auslösers, wo wir ja vorher nur den HINT-Befehl stehen hatten. Das bedeutet, im Auslöser steht jetzt:

[2] Wie so oft ist dies nur die halbe Wahrheit. Bestimmte Dateien wie die description.ext erfordern z.B. das einmalige Speichern, bevor Änderungen übernommen werden können. Auch werden bereits laufende Skripte logischerweise nicht mehr verändert, das betrifft vor allem die Init.sqf. Eine Änderung in der mission.sqm muss durch Laden der Mission im Editor sichtbar gemacht werden.

[3] Wir verweisen an dieser Stelle direkt auf das Kapitel I. 4.B. Da wir hier meist Befehle in einem beliebigen Kontext betrachten, ist zu erwarten, dass der Leser unsere Befehle nicht alleine nutzt, daher sind alle Codezeilen stets mit „;“ am Ende.



Sofern sich `meinScript.sqf` im Hauptverzeichnis eurer Mission befindet, sollte die Vorschau wieder ein brauchbares Ergebnis liefern. Solltet ihr den Befehl aber aus diesem Handbuch einfach kopiert haben, so werdet ihr eine Fehlermeldung erhalten:

Dieser Fehler rührt daher, dass ihr die Anführungszeichen mitkopiert habt, diese aber eine andere Schriftart darstellen und als Sonderzeichen eingefügt werden – sprich: Das Spiel erkennt sie schlichtweg nicht. Daher solltet ihr Befehle mit Vorsicht kopieren, bei Standard-Schriftarten wie Arial gibt es selten Probleme. Ihr könnt daher die Befehle aus dem Comref oder dem BI Wiki ohne Probleme kopieren.

Wir haben also drei Wege kennengelernt, um Scripte in das Spiel zu integrieren, auch wenn sie bisher noch nicht sehr viel können, aber das kommt noch. Wir können die Init-Zeile, die Aktivierungs- oder Deaktivierungszeile sowie ein externes Script benutzen, das wir dann über EXECVM aufrufen müssen.

Im nächsten Abschnitt wollen wir einen kurzen Blick auf die Fehlerbehandlung und die ACE-Debug-Konsole werfen, ein Hilfstool, das euch unglaublich viel Zeit und Arbeit abnehmen kann!

I. 3. FEHLERMELDUNGEN UND FEHLERSUCHE

A FEHLERMELDUNGEN IM SPIEL

Wir sind bisher auf mindestens eine Fehlermeldung gestoßen (s. S. 15). ArmA verfügt also bereits über eine rudimentäre Fehlerengine, die uns über grobe Schnitzer und Patzer aufklärt. Dies passiert immer dann, wenn wir unvollständige Befehle oder Befehle mit falscher Syntax in die Initzeile eines Objektes oder in die drei Felder eines Auslösers schreiben.

Uns interessieren aber vor allem Fehler in unseren Scripten, also Fehler, die erst zur Laufzeit des Scriptes auftreten und die schlimmstenfalls nie angezeigt werden. Zunächst wollen wir uns aber mit dem Thema der Fehleranzeige beschäftigen, d.h. welche Möglichkeiten bietet ArmA2 auf Fehler hinzuweisen.

Neben der oben beschriebenen Fehlermeldung besitzt ArmA einen extrem wichtigen Startparameter, der Scriptfehler bei Auftritt anzeigt. Dieser Startparameter lautet **-showScriptErrors** und wird je nach Startmethode entweder ans Ende eurer Verknüpfung (Rechtsklick >> Eigenschaften >> Zielzeile) oder z.B. bei Yoma[4] unter dem Reiter Erweitert >> Verschiedene Argumente dazugeschrieben.

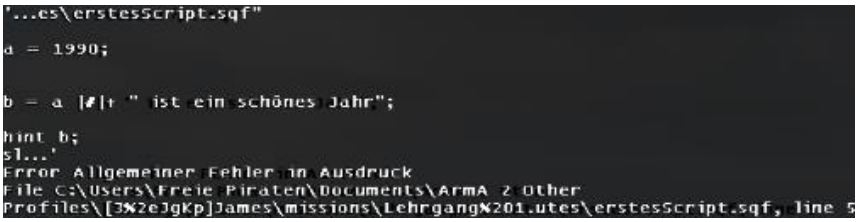
Haben wir das Spiel mit diesem Parameter gestartet, so bekommen wir im Falle eines Scriptfehlers stets eine schwarze Hinweismeldung relativ mittig im Bildschirm. Dabei ist es egal, ob der Scriptfehler in der Scriptzeile eines Objektes oder in einem externen Script passiert ist, der Debugger findet beide!

Ihr könnt direkt den Debugger nach dem Starten des Editors testen, indem ihr unsere HINT-Meldung auf verschiedene Weisen mit Absicht falsch eingebt. Zum Beispiel solltet ihr einmal statt HINT nur hin eingeben, den Text ohne Anführungszeichen schreiben oder Text und HINT-Befehl in der Reihenfolge vertauschen. Ergebnis wird sein:

Mit etwas gutem Willen ist die Meldung gut zu lesen. In den ersten Zeilen steht immer der Fundort des Fehlers. Die Meldung kann auch bei längeren Scripten deutlich länger ausfallen, das sollte klar sein. Wichtig ist immer die Zeile mit „Error“, denn hier taucht die Fehlermeldung auf: „Fehlendes ;“, sowie das kleine Rautensymbol |#|, das euch auf die Stelle des Fehlers hinweist.

In diesem Fall haben wir das Script direkt im Editor in die Init-Zeile eingegeben. Im ersten Fall haben wir statt HINT den Befehl ohne t eingegeben, was natürlich zu einem Fehler führt. Da er den Befehl nicht mehr erkennt, kann er auch nicht auf das fehlende t hinweisen, sondern merkt nur, dass der Befehl nicht korrekt mit einem Semikolon beendet wurde. Im zweiten Fall haben wir Text und Befehl getauscht, der Fehler tritt diesmal **vor** dem HINT–Befehl auf, da ein STRING nicht alleine stehen darf, wir haben die Syntax verletzt.

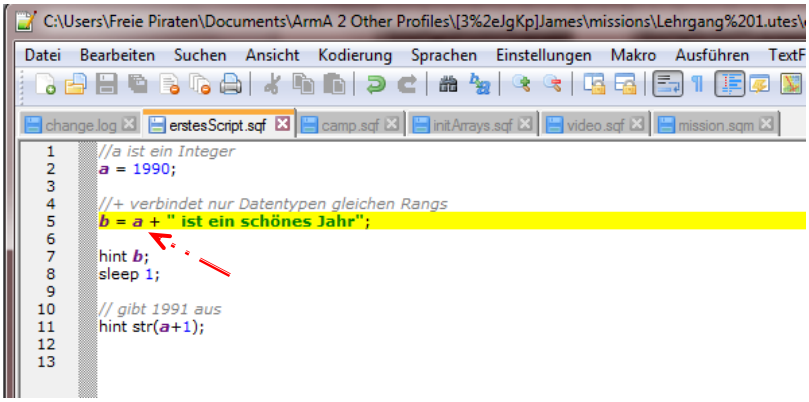
Jetzt haben wir noch die Möglichkeit, dass der Fehler im externen Script auftaucht. Auch dies zeigt uns der Debugger an, jedoch mit etwas mehr hilfreichen Informationen:



Nicht erschrecken – zu Demonstrationszwecken haben wir ein anderes Script genommen, das etwas mehr Zeilen enthält. Der Inhalt ist überhaupt nicht wichtig, auch die Ursache des Fehlers soll uns nicht interessieren, es geht alleine um das Lesen der Meldung! Wir sehen, dass zunächst der Code erscheint, bis die Error-Zeile auftritt, die einen „Allgemeine[n] Fehler in Ausdruck“ berichtet. Danach erscheint bei externen Scripten

[4] http://www.yomatools.be/

immer die Pfadangabe der Datei und – ganz wichtig – die Zeilennummer! Diese also unbedingt einblenden lassen, falls euer Editor so etwas unterstützt. Zudem sehen wir auch wieder das |#|-Symbol. Damit sind wir in der Lage, den Fehler exakt im Script zu orten:



Dies ist Notepad++ mit installierter SQF-Syntaxhervorhebung und aktivierten Zeilennummern. Damit können wir direkt die Fehlermeldung 1:1 nachverfolgen und wissen, dass der Fehler in der markierten Zeile Nummer fünf bei der Variablen a liegen muss. Warum der Fehler auftritt und was wir hier mit dem Script bezwecken, ist nebensächlich, mitnehmen sollt ihr nur, wie die Meldung zu lesen ist, so dass ihr in euren Scripten die Fehler findet.

B DIE DEBUG-KONSOLE DES EDITORS

Bisher war die Fehlersuche relativ einfach. Wir hatten kurze Scripte mit wenigen Zeilen und Befehlen und der Debugger war in der Lage, den Ort des Fehlers relativ zielsicher anzugeben, so dass einer Behebung eigentlich nichts im Wege stand. Es gibt aber – leider häufiger als man denkt – Situationen, in denen kann sich die Suche nach einem Fehler als sehr tückisch erweisen. Das sind vor allem solche Fälle, wo gar kein Fehler vorliegt, sondern wo wir einen Logik-Fehler begangen haben. Scriptfehler äußern sich zu 99 % in einer Fehlermeldung, weil der Debugger bzw. der Compiler weiß, wie die Syntax aussehen muss und z.B. ein fehlendes Semikolon immer bemerkt, da es seiner Funktionsweise entspricht, Anweisungen nach jedem Semikolon auszuführen. Logik-Fehler sind jedoch Fehler, bei denen sich ein Programm nicht so verhält, wie wir es ursprünglich beabsichtigt hatten, weil wir z.B. einen Befehl falsch verstanden haben. Diese Art von Fehlern resultiert meistens in einer unerwünschten Äußerung oder in einem Ausbleiben einer Aktion. So könnten wir den HINT-Befehl auch mal mit einer Zahl probieren:

```
hint 5;
```

Das ist unser ganzes Script! Bitte probiert dies mit einem externen Script, dass ihr im Spiel über EXECVM ladet. Wir erhalten keinen Logik-, sondern einen Script-Fehler, da wir die Syntax verletzt haben. HINT erwartet einen STRING, wir liefern aber eine Zahl (NUMBER, später mehr). Jetzt könnten wir die 5 in Anführungszeichen setzen und hätten den Fehler behoben. Wollten wir jetzt eine Addition ausgeben (weil wir aus dem Matheunterricht die Addition als eines der wenigen Dinge behalten haben, die recht nützlich sind), so würde die Zeile

```
hint 5+5;
```

erneut denselben Fehler verursachen, also probieren wir dieselbe Lösung:

```
hint "5+5";
```

Sieht logisch aus, ist korrektes „SQF“ und führt zu einer…nicht erwarteten Ausgabe, nämlich „5+5“. Wir haben einen Logik-Fehler[5] begangen, denn wir geben mit dem HINT-Befehl immer einen Text aus, und die Addition als Text ist genau das, was wir als Ergebnis erhalten. Wir wollten eigentlich zunächst eine

[5] Man könnte auch von falschen Datentypen reden, siehe Kapitel 0

Berechnung ausführen und das Ergebnis als Ausgabe produzieren, dem Computer gesagt haben wir das aber nicht.

Auch wenn wir noch keine Datentypen behandelt haben, so möchten wir an dieser Stelle etwas vorgreifen, um die Debug-Konsole, von der bisher noch nicht weiter die Rede war, sinnvoll einführen zu können. Daher bitten wir um etwas Nachsicht, wenn nicht alle Details des jetzt noch folgenden Abschnitts sofort klar verständlich sind. Es geht – wie immer – um das Prinzip, wie ihr diese Konsole nutzen könnt.

Um also Logik-Fehlern auf die Schliche zu kommen, bedient man sich in der Programmierung meistens sogenannter Tracer-Tabellen. Dabei macht man nichts anderes, als sich eine kleine Tabelle zu schreiben, die für jeden Moment (sozusagen jede Zeile des Scriptes) den aktuellen Wert der interessierenden Variablen festhält. Versteht man nicht? Macht nichts, hier ein Bild:

	a	b	t
int a, b;	*undefiniert*	*undefiniert*	
a = 1234;	1234	*undefiniert*	
b = 99;	1234	99	
int t = a;	1234	99	1234
a = b;	99	99	1234
b = t;	99	1234	1234

Ihre erste Trace-Tabelle

Dieses Beispiel stammt aus JAVA, soll euch aber nicht verwirren. Die Variablen a und b sind einfach zwei Schubläden, die Werte enthalten. a wird mit 1234 und b mit 99 angelegt, diese Werte sind also zuerst in den Schubläden. Danach wird eine dritte Schublade geholt, die wir t nennen, und die ebenfalls den Wert aus a erhält. Anschließend schütten wir den Inhalt von b nach a (alles ohne wirklich zu schütten, was in b ist, bleibt in b). Zum Schluss wird noch einmal der Inhalt aus der Schublade t in die Schublade b geschrieben. Damit haben wir die Inhalte bunt durcheinandergemischt. Worum es uns bei diesem Beispiel geht, ist der Sinn der Tabelle. In SQF würde der Code wie folgt lauten:

Listing_1.sqf

```
1  a = 1234;    // Variable a wird initialisiert
2  b = 99;      // Variable b wird initialisiert
3  t = a;       // Variable t wird mit a initialisier
4  a = b;       // Variable a wird neu mit b belegt
5  b = t        // Variable b wird neu mit t belegt
```

Wir müssen in SQF also keine Variablen vorher „deklarieren“, aber um solche Feinheiten soll es noch nicht gehen. Wenn wir diesen Code als externes Script angeben und im Spiel aufrufen, haben wir drei Werte (Variablen), die wir uns ausgeben lassen könnten, nämlich a, b und t. Jetzt ist es aber sehr umständlich, dafür jedes Mal mit HINT zu arbeiten (und noch können wir damit nur Text ausgeben)! Die Lösung bietet die Debug-Konsole, die wir jetzt öffnen wollen. Also führt zunächst das Script im Editor aus, drückt dann Esc.

Die Debug-Konsole[6] befindet sich dann ganz unten im Menü. Sie besteht aus zwei wesentlichen Teilen

[6] Momentan ist diese Konsole ein ACE-Feature, wer also ohne ACE spielt, wird ohne die Konsole auskommen müssen. Wir arbeiten aber bereits an einer Standalone-Version, dann wird es ein Update dieses Passus‘ geben.

Im oberen Teil sehen wir nur leere Zeilen. Dies ist der sogenannte Tracer-Bereich. Er funktioniert wirklich einfach: Man schreibt den zu überwachenden Ausdruck in die erste Zeile und die Konsole spuckt einem den Wert in der zweiten Zeile aus. Großer Vorteil: Das passiert vollautomatisch und permanent aktualisiert. Ihr könnt ja mal in die Zeile den Befehl TIME schreiben.

Um unser Beispiel weiter fortzuführen, haben wir im obigen Screenshot bereits die drei Werte a, b und t eingegeben. Natürlich ist dies keine richtige Tracer-Tabelle, da wir jetzt nur die Endwerte abfragen, aber es ist trotzdem ein sehr großer Vorteil und eine Zeitersparnis gegenüber der manuellen Ausgabemethode. Ihr könnt also jede beliebige Variable, die ihr in Zukunft braucht, hier abfragen. Das betrifft sowohl eure eigenen Scripte als auch Variablen, die Objekte oder Zustände betreffen. So kann die Spielzeit z.B. mit DATE abgefragt werden. Damit habt ihr eine einfache Möglichkeit, eine gesuchte Größe zu überprüfen und z.B. gesuchte Werte zu finden.

Der zweite Bereich ist sozusagen eine Entwicklungsumgebung für Scripte. Alles, was ihr extern scriptet oder in eine Init-Zeile schreiben wölltet, könnt ihr hier 1:1 ausprobieren und direkt ausführen. Das heißt, ihr braucht nicht länger z.B. in Auslöser laufen, um eure Scripte zu testen! Ihr könnt entweder eigene Scripte via EXECVM direkt ausführen oder Teile davon als eigenständigen Code eingeben. Dieser muss dann aber irgendeine Ausgabe oder einen Effekt produzieren, um sichtbar zu werden. Ihr werdet von der Konsole dabei mit einer Autovervollständigung unterstützt, die bei jeder Eingabe sofort aktiviert wird. Den angezeigten Befehl übernehmt ihr mit ⇥.

Habt ihr also euren Code eingegeben, könnt ihr rechts auf den Button „Exec“ drücken und der Code wird sofort ausgeführt. Handelt es sich z.B. um einen HINT-Befehl, erscheint auch direkt die Ausgabe. Habt ihr irgendeinen Fehler fabriziert, wird euch dies ebenfalls in der besprochenen Weise angezeigt, sofern ihr –SHOWSCRIPTERRORS aktiviert habt. So könnt ihr also z.B. einen fehlerhaften Code aus eurem Script in die Debug-Konsole kopieren und solange korrigieren, bis kein Fehler mehr angezeigt wird. Das erspart euch ein permanentes Wechseln zwischen Desktop und Spiel!

Übungen

[1] Wiederholt für euch noch einmal alle drei gezeigten Methoden des Scriptaufrufs bzw. der Scripteingabe. Welche waren das?

[2] Lernt das Comref und das BI Wiki zu lesen. Dazu versucht, folgenden Befehl zunächst zu finden und seine Eigenschaften zu erfahren. Danach soll er im Editor oder via Skript benutzt werden: **random**.

[3] Versucht den Befehl aus Aufgabeteil 2 mit der Debug-Konsole „auszuwerten“. Dazu solltet ihr den Befehl in eine Tracer-Zeile schreiben und herumprobieren. Da der Befehl einen Wert liefert, klappt dies auch.
Was fällt euch bei besagtem Befehl auf? Werden alle Werte angenommen, die ihr erwartet? Ihr könnt dies erst einmal für den Fall zwischen 0 und 1 testen.
In der Tat erzeugt dieser Befehl (Wiki lesen!) nur Werte zwischen einschließlich 0 und ausschließlich eurem Wert. Was müsste man theoretisch tun, um den obersten Wert zu erhalten? Wir werden später eine Lösung dafür präsentieren.

I. 4. DER GRUNDLEGENDE AUFBAU EINER SQF-DATEI

Glückwunsch! Wer bis hierher gekommen ist, hat zumindest als Anfänger eine große Hürde genommen. Ihr seid mit uns immerhin mitten ins kalte Wasser gesprungen und habt bereits allerhand über Fehler lernen müssen, ohne immer im Detail folgen zu können (was wir annehmen, was aber nicht stimmen muss). Damit wir jetzt wirklich Jeden mitnehmen und niemand auf der Strecke bleibt, werden wir mit den versprochenen Grundlagen beginnen. Die folgenden Abschnitte mögen daher zunächst etwas trocken wirken, legen aber das Fundament für alles, was kommt und für die Scriptarbeit im Gesamten.

Wir werden uns mit der Syntax, den Datentypen, Kontrollstrukturen und Feinheiten von ArmA beschäftigen.

A SYNTAX

SQF ist – Gott sei Dank – relativ benutzerfreundlich, was die Eingabe von Befehlen angeht. Andere Sprachen erfordern vor der ersten wirklichen Anweisung zum Beispiel Klassendefinitionen und Methodenrümpfe. In SQF haben wir all das zunächst nicht. Wir wollen nicht verschweigen, dass bestimmte Dateien wie SQM, EXT und HPP-Dateien eine Klassenstruktur aufweisen, für uns spielt dies aber noch lange keine Rolle.

Der obige Absatz soll also besagen: Wir können prinzipiell fast alle Befehle und Anweisungen ohne Bedenken einfach so „abschreiben“ bzw. benutzen, ohne vorher irgendwelche Definitionen einführen zu müssen. Wir müssen lediglich die Regeln der Sprache beachten (siehe obige Definition: S. 11). Erinnern wir uns an unser erstes Beispiel aus Kapitel I. 2., so bestand unser Script aus einer einzigen Zeile, die wir im Grunde genau so aus der Skriptreferenz entnommen haben. Das einzige, was wir uns in SQF prinzipiell merken müssen, ist:

JEDER AUSDRUCK ODER BEFEHL WIRD PRINZIPIELL MIT EINEM SEMIKOLON BEENDET!

Missachten wir diese einfache Regel, wird der Compiler in 99% aller Fälle eine Fehlermeldung anzeigen, die zum Glück genau das besagt: Es fehlt ein Semikolon (haben wir bereits kennengelernt). Es gibt eine Ausnahme von dieser Regel: Die letzte Zeile eines Scriptes braucht kein Semikolon als Abschluss. Dies ist aber so speziell, dass ich von der Anwendung abrate! Wer sich näher hierrüber informieren will: Kapitel I. 4.B.

Ein Ausdruck ist ein isoliertes, eigenständiges Code-Fragment, das einen bestimmten Wert repräsentiert oder ausgibt. Ausdrücke können Variablen, Operationen (ohne Zuweisung!) oder Skriptbefehle mit Rückgabewert sein. Ausdrücke dürfen nicht mit Befehlen verwechselt werden:

EIN BEFEHL (STATEMENT) IST EIN CODE-FRAGMENT, DAS EINE ABSICHT DARSTELLT UND EINE AUSGABE PRODUZIERT.

Befehle können Initialisierungen (Wertzuweisungen), Kontrollstrukturen (Bedingungen/Schleifen) oder Befehle sein.

Als zweitwichtigstes Element in Scripten brauchen wir Kommentare. Kommentare sind für uns die Gedankenstütze, die uns erlaubt, unser eigenes Script nachzuvollziehen und unsere Entscheidungen auch auf längere Sicht festzuhalten, damit wir später nicht wieder rätseln müssen, warum wir uns für diese Variable, jenen Wert oder diese Lösung entschieden haben. Kommentare haben aber ihren eigentlichen Wert für denjenigen, der euer Script eines Tages einmal erweitern oder nachvollziehen möchte. Wer sich einmal ein Script z.B. von http://www.armaholic.com/index.php heruntergeladen hat und am Inhalt verzweifelte, weil er überhaupt nichts verstanden hat, wird wissen, wovon wir reden. Jeder Programmierer (und dazu zählen wir uns ja!) sollte selbst bei seinem allerersten Skript Kommentare benutzen. Wir raten euch, solange ihr „unerfahren“ seid, jeden neuen Befehl als Kommentar mit seiner Syntax aus dem Wiki abzuschreiben und so jederzeit in eurem Skript als Minireferenz verfügbar zu halten.

KOMMENTARE IN SQF WERDEN DURCH // EINGEFÜGT. DIESE KOMMENTARE SIND ZEILENKOMMENTARE.

LÄNGERE KOMMENTARE ÜBER MEHR ALS EINE ZEILE KÖNNEN MIT /...*/ EINGEGEBEN WERDEN.*

Wir haben also zwei Möglichkeiten, unseren Code entsprechend zu kommentieren. Meistens reichen uns Doppelschrägstriche völlig aus.

Als Letztes gibt es noch Blöcke.

Blöcke gruppieren zusammengehörigen Code als logische Einheit und sind in {...} eingeschlossen.

Blöcke ergeben sich automatisch bei der Verwendung von Kontrollstrukturen und Schleifen oder bei den Befehlen CALL und SPAWN, denen wir aber erst viel später begegnen werden (also den letzteren).

Damit könnte eine zunächst sinnfreie, wenn auch korrekte SQF-Datei so aufgebaut sein:

Listing_2.sqf

```
1  // Dies ist ein Skript und tut Folgendes…
2  erste Anweisung;
3  zweite Anweisung;
4
5  // Jetzt folgt ein Block, also eine logische Code-Einheit
6  Kontrollstruktur
7  {
8    Anweisung innerhalb des Blocks;
9  ...
10 }; // Ende des Blocks aber nicht unbedingt des Skriptes, daher ein ;
```

Wir finden in diesem Beispiel die Elemente Kommentare, Anweisung und Block wieder. Wir haben jede Anweisung mit einem Semikolon beendet. Auch ein Block muss am Ende der geschweiften Klammer stets mit einem Semikolon beendet werden (im Gegensatz zu den meisten anderen Scriptsprachen wie Java). Wir könnten den selben Code auch wie folgt schreiben:

```
erste Anweisung;zweite Anweisung;Kontrollstruktur{Anweisung innerhalb des
Blocks;...};
```

Damit hätten wir doch deutlich Platz gespart. Leider halten wir diese Schreibweise weder für ästhetisch (das wäre verkraftbar) noch für lesbar und damit – das ist das Entscheidende – wartbar. Es gibt in jeder Programmiersprache sogenannte Konventionen, die sich durchgesetzt haben. Die wichtigsten dieser ungeschriebenen Regeln sind, dass jede Anweisung in einer eigenen Zeile steht, dass umfangreicher Code kommentiert werden sollte und dass verschachtelte Anweisungen mit einer Einrückung abgehoben werden. Gerade bei späteren Scripten, die mehrere Schleifen oder Bedingungen aufweisen, ist es sehr wichtig, die Übersicht zu behalten. Und Übersicht bedeutet beim Programmieren fast immer gleichzeitig eine saubere Notation mit Mut zur „Lücke“, also Freiraum, Absätze und Tabstopps. Der Grund, warum so etwas funktioniert, liegt an der Funktionsweise des Compilers/Interpreters, die im Hintergrund quasi euer Script in Maschinencode übersetzen. Denn alles, was wir per Editor schreiben, ist natürlich nur „Pseudo-Code“ und für den Rechner völlig unverständlich. Daher muss ein Stück Software euer Script übersetzen, damit ArmA mit euren Anweisungen überhaupt etwas anfange kann. Dieses Übersetzen beinhaltet z.B. auch eine sehr starke Bereinigung des Codes von allem Ballast, den wir für die bessere Lesbarkeit einfügen, den ein Computer aber schlichtweg nicht braucht. Leerzeichen, Kommentare, Einrückungen, Absätze: All das sind unnötige Details, die vom Compiler entfernt werden. Daher könnt ihr euer Script so ausgestalten, wie es euch gefällt, es wird im Spiel keinen Unterschied – auch nicht in der Performance – bewirken.

Ein letztes Wort zu Variablen: Auch wenn wir an dieser Stelle noch keinerlei Befehle (bis auf den Einstieg) kennengelernt oder Datentypen behandelt haben, so wird jedem klar sein, dass wir für Skripte Variablen brauchen, die unsere Berechnungen oder allgemein Werte speichern. Gemäß dem Prinzip der Lazy evaluation sollten wir einfach darauf achten, so wenig (Hilfs-)Variablen wie nötig einzuführen und nur dort Variablen wirklich zu gebrauchen, wo wir sie auch verwenden wollen. Ausdrücke/Berechnungen können manchmal besser einfach so genutzt werden, als erst gespeichert werden zu müssen (was Speicherplatz kostet) und manche Variable kann man getrost weglassen, da sie am Ende doch nicht gebraucht wird.

B WO WIR SEMIKOLA BRAUCHEN UND WO NICHT – SOWIE EINIGE GEDANKEN ZU KLAMMERN

In diesem Kapitel wollen wir einen soliden Grundstein für die weitere Arbeit mit Skriptbefehlen legen. Dazu bedarf es noch zweier Anmerkungen zu den oben angesprochenen Konventionen. Beide können leider von kompletten Neueinsteigern noch nicht zur Gänze an dieser Stelle nachvollzogen werden, wir bitten daher um Entschuldigung, aber das geniale an Online-Dokumentationen respektive PDFs ist ja die Möglichkeit, jederzeit via Navigationsleiste hierher zurückzuspringen. Also stürzen wir uns in die Details!

Wir erinnern uns an die goldene Regel aus dem vorherigen Kapitel: Jede Anweisung wird durch ein Semikolon beendet. Dies können wir prinzipiell so stehen lassen. Außerdem hatten wir bereits eine kleine Ausnahme von dieser Regel beschrieben: Eine letzte Anweisung innerhalb eines Skriptes bedarf keines Semikolons, da wir hier kein weiteres Trennzeichen zwischen mehreren Anweisungen brauchen. Das Semikolon ist für den Compiler nichts weiter als ein logischer Separator, um Anweisungen klar voneinander trennen zu können. Daher ist bei der letzten Anweisung ohne Semikolon auch für den Compiler klar, dass hier nichts mehr kommt. Wir hatten dies oben noch als Spezialfall abgetan, doch an dieser Stelle möchten wir ein bis zwei Beispiele zeigen, wo dieses eine kleine Semikolon noch deutlich öfter weggelassen werden kann, nämlich immer dann, wenn wir Verschachtelungen vor uns haben.

Doch zunächst noch einmal zum einfachsten Fall, damit wir niemanden zurücklassen:

```
Anweisung1;
Anweisung2;
letzteAnweisung
```

In diesem wirklich sehr einfachen (und funktionslosen, aber das ist egal) Beispiel ist die letzte Anweisung ohne abschließendes Semikolon, da wir ein Semikolon – wie gesagt – nur dann brauchen, wenn weitere Anweisungen folgen.

Dasselbe gilt natürlich auch, wenn wir innerhalb von Code-Blöcken die letzte Anweisung schreiben:

Listing_3.sqf

```
1  if (Bedingung == Prüfwert) then
2  {
3    Anweisung1;
4    Anweisung2
5  }
```

Hier müssen wir jetzt schon zwei Dinge beachten: Zum einen gilt weiterhin, dass `Anweisung2` ohne Semikolon geschrieben werden kann, weil sie die letzte Anweisung innerhalb des Blockes ist. Was eine IF-Abfrage ist, muss im Moment noch nicht bekannt sein, es ist auch nicht wichtig. Wichtig ist die zweite Erkenntnis: Auch die abschließende geschweifte Klammer (die den Block schließt) steht ohne Semikolon! Wir haben also im Vergleich zur allgemeinen Regel zwei Semikola eingespart, weil nach `Anweisung2` keine weitere Anweisung kommt, und die schließende Klammer zur öffnenden gehört.

Nach diesem Prinzip können wir dann auch das letzte Beispiel besprechen. Ich danke an dieser Stelle Vienna für die Bereitstellung desselben. Wir zeigen zunächst den Code, wie er nicht geschrieben werden sollte (mit roten Hervorhebungen). Als zweites folgt dann das korrekte Beispiel.

Aber eines vorweg: Korrekt und nicht korrekt sind starke Worte, denn syntaktisch ist beides in Ordnung. Ein Semikolon zu viel ist kein grammatikalischer Fehler, also nichts, worüber ein Compiler meckern würde. Es sieht zum Teil einfach ohne Semikola besser aus und zum anderen ist die Reihenfolge mit richtigen Semikola klarer, da Semikola eben weitere Anweisungen implizieren. Aber für den Anfänger sollte immer gelten: Bitte eignet euch zuerst die wichtigen Regeln zu 100% an. Wenn ihr dann die Feinheiten verstanden habt und wisst, wann ihr warum Dinge vereinfachen oder weglassen könnt, dann ist dies in Ordnung. Wer aber unsicher ist, oder plötzlich vom Compiler Fehler angezeigt bekommt, sollte sicherheitshalber bei der Regel

bleiben, dass hinter jede Zeile (was bei uns mit Anweisung identisch sein sollte) ein Semikolon gehört. Nun aber zum eigentlichen Code-Beispiel. Zunächst mit allen Semikola, ob notwendig oder nicht.

Listing_4a.sqf

```
temp = [Flugobjekt,Absprungintervall] spawn
{
  private ["_flugobjekt","_intervall","_units","_springer"];
  _flugobjekt = _this select 0;
  _intervall  = _this select 1;
  if (local driver _flugobjekt) then
  {
    _units = crew _flugobjekt;
    for "_i" from 0 to count _units - 1 do
    {
      _springer = _units select _i;
      if (_springer != driver _flugobjekt) then
      {
        _springer action ["EJECT",_flugobjekt];
        sleep _intervall;
      };
    };
  };
};
```

Im zweiten Listing betrachten wir das selbe Beispiel, diesmal mit korrekter Semikola-Verwendung.

Listing_4b.sqf

```
temp = [Flugobjekt,Absprungintervall] spawn
{
  private ["_flugobjekt","_intervall","_units","_springer"];
  _flugobjekt = _this select 0;
  _intervall  = _this select 1;
  if (local driver _flugobjekt) then
  {
    _units = crew _flugobjekt;
    for "_i" from 0 to count _units - 1 do
    {
      _springer = _units select _i;
      if (_springer != driver _flugobjekt) then
      {
        _springer action ["EJECT",_flugobjekt];
        sleep _intervall
      }
    }
  }
};
```

Auch wenn Viele jetzt vielleicht abgeschreckt sind, wir wiederholen es noch einmal: Der Sinn und Zweck dieses Skriptes ist unwichtig (sollte aber nach dem Lesen von Kapitel I. 6 keine Probleme mehr bereiten). Es kommt nur auf die Verwendung der Semikola an. Im Vergleich zum letzten Beispiel kommt aber keine neue Erkenntnis hinzu, lediglich die Verschachtelungen sind zahlreicher und deshalb können ab der innersten IF-Abfrage alle nachfolgenden Semikola weggelassen werden. Sobald man jedoch nach einer der Schleifen oder Verzweigungen eine weitere Anweisung braucht, muss natürlich auch hinter eine der inneren abschließenden geschweiften Klammern ein Semikolon gesetzt werden. Selbst das äußerste Semikolon (roter Kasten) kann weggelassen werden, wenn das Skript danach beendet sein sollte.

Damit haben wir wahrlich genügend Worte zu diesem Thema verloren und möchten stattdessen noch einige Hinweise zur Verwendung von Klammern geben. Klammern sind grundsätzlich rein optional – das bedeutet, ihr könnt Klammern nach eigenem Gutdünken setzen. Es gibt davon grundsätzlich nur zwei Ausnahmen: Klammern legen die Berechnungsreihenfolge und bei verschachtelten Befehlen die Zusammengehörigkeit fest.

Prinzipiell ist es euch also zunächst freigestellt, welche Variante ihr bevorzugt:

```
Array = [5, "ein String", 3+4, (1+2)];
```

Obwohl wir ARRAYs erst später behandeln werden, geht es hier lediglich darum, dass die 4 Einträge innerhalb der eckigen Klammern jeweils ein Element darstellen. Bei der `5` und dem Text innerhalb der Anführungszeichen ist dies offensichtlich, bei den beiden letzten Einträgen handelt es sich aber erst einmal um Code, der ausgeführt werden muss, ehe der Wert eingetragen werden kann. Dies macht der Compiler aber *on the fly* – also direkt zur Laufzeit. Hier geben wir zwei Tipps: Verzichtet auf Klammern, wenn diese die Lesbarkeit eher behindern als fördern und verzichtet auf Hilfsvariablen, wenn sie nicht gebraucht werden. So sind die Einträge `3+4` und `1+2` gleichwertig, die Klammern im letzten Fall heben nur für den menschlichen Leser hervor, dass hier zunächst eine Berechnung erfolgen soll, ehe das Ergebnis (3) dann als vierter Eintrag übernommen werden soll. Außerdem ist es nicht notwendig, immer jede Berechnung zwischenzuspeichern, wenn sie sonst nicht mehr gebraucht wird, das kostet nur Speicherplatz und Performance:

Listing_5.sqf

```
1  _ersteZahl = 1;
2  _zweiteZahl = 2;
3  _summe = _ersteZahl + _zweiteZahl;
4  hint str(_summe) // gleich mit: hint str(1+2)
```

Auch wenn dieses Beispiel wahrlich banal ist, so ist das Banale oft am besten geeignet, sich gewisse Konzepte und Prinzipien klarzumachen. In diesem Falle sollte es offensichtlich sein, dass die Hilfsvariablen `_ersteZahl` und `_zweiteZahl` hier nicht sinnvoll sind, wenn es lediglich um die Ausgabe des Additionsergebnisses geht. Als Faustregel sollten Werte immer nur dann in Variablen gespeichert werden, wenn wir sie wirklich mehrfach im Skript brauchen oder wenn ihre Berechnung bzw. ihre Ermittlung zu viel Platz in Anspruch nimmt, als das es noch sinnvoll händelbar wäre. Gerade bei ARRAYs wird dies oft der Fall sein:

Listing_6.sqf

```
1  _zweiterEintrag = array select 1; // klare Benennung hilft beim Umgang
2  _mittelwert = (zahl1+zahl2) / 2; //Hilfsvariable ist einfacher als Berechnung
```

Diese zwei Beispiele kämen auch ohne eine Speicherung in einer Variable aus, wir könnten also auch `array select 1` so benutzen bzw. den Mittelwert der zwei Zahlen direkt bilden und weiterverarbeiten. Gerade bei komplexeren Skripten ist es aber sehr von Vorteil, mit aussagekräftigen Namen zu arbeiten, deren Inhalt sich direkt durch das Lesen erschließt. Daher gilt durchaus, dass eine sinnvoll benannte Variable mehr wert ist als immer um jeden Preis auf Variablen zu verzichten.

Im zweiten Beispiel – dem Mittelwert – sehen wir auch die eigentliche Verwendung von Klammern: Sie sollen dem Compiler mitteilen, welche Rechenschritte bzw. Bearbeitungsschritte auf jeden Fall zuerst auszuführen sind. Jedem sollte einleuchten, dass die Klammern hier absolut notwendig sind, um das zu erhalten, was gesucht ist. Ohne Klammern würden wir in Wahrheit `zahl1` plus die Hälfte von `zahl2` rechnen. Dies gilt aber eben nicht nur bei diesen leicht nachzuvollziehenden mathematischen Berechnungen, Klammern kommen auch bei mehreren SQF-Anweisungen in Folge zum Einsatz, weil die Intelligenz des Compilers irgendwann an ihre Grenzen stößt:

```
_eineZahl = ceil random count units group player - 1;
```

Obwohl einige Befehle noch nicht bekannt sein dürften, wollen wir an dieser Stelle dieses Beispiel kurz erläutern, um die Bedeutung von Klammern hervorzuheben: Wir wollen eine Zufallszahl ausgeben, die irgendwo zwischen 1 und der Anzahl der Gruppenmitglieder des Spielers minus Eins ist. Zunächst brauchen wir den Befehl COUNT, der die Einheiten zählt. Danach übergeben wir diese Zahl dem Befehl RANDOM, der eine Zahl zwischen 0 und der übergebenen als Fließpunktzahl zufällig ausgibt. Da wir 0 ausschließen wollen, runden wir die Zahl immer nach oben, das geschieht mittels **ceil**.[7] Als letztes reduzieren wir die so erhaltene Zahl um eins. Hier geht es weniger um den unmittelbaren Sinn als um die heroenhafte Leistung, die der Compiler erbringen muss. Er muss nämlich bei jedem Befehl erkennen, welcher Parameter erforderlich ist und wie dieser berechnet wird! Da dies gerade bei solch komplexen Abfolgen mehrere Befehle oft nicht zum gewünschten Ergebnis oder gar zu Scriptfehlern führt, wäre eine Möglichkeit, alle logisch sinnvollen Klammern zu setzen:

```
_eineZahl = (ceil ( random (count (units group player)))) - 1;
```

Diese Variante ist narrensicher – nur nicht jedermanns Geschmacksache. Wir geben hier aber keine bindenden Konventionen oder gar Stilregeln vor, daher möchten wir es bei diesem letzten Beispiel bewenden lassen. Jeder soll sich selbst für eine Variante entscheiden, die für ihn das Optimum zwischen Lesbarkeit und korrekter Syntax darstellt. Sicherlich wird dieses irgendwo in der Mitte liegen.

Leider gibt es keine festen Regeln, wann Klammern notwendig sind und wann nicht. Manchmal klappt ein solch langer Befehl wie der obige ohne eine einzige Klammer, manchmal liefert der Compiler einen Fehler. Als Faustregel gilt hier, dass keine Klammern notwendig sind, wenn die Befehle von innen nach außen direkt hintereinander folgen, sich also quasi immer auf den weiter links stehenden Befehl beziehen. Bei unserem Beispiel ist die `-1` das Problem, da diese sich auf CEIL beziehen soll. Dazu sind Klammern notwendig. Alternativ könnten wir natürlich das Ganze in kleinen Schritten angehen:

Listing_7.sqf

```
1  _personen = count units group player; //Anzahl der Personen feststellen
2  _auswahl  = random _personen;         //eine der Personen zufällig auswählen
3  _auswahl  = ceil _auswahl;            //umwandeln auf Ganzahl ab 1
4   oder
5  _auswahl  = floor _auswahl;           //umwandeln auf Ganzahl ab 0
```

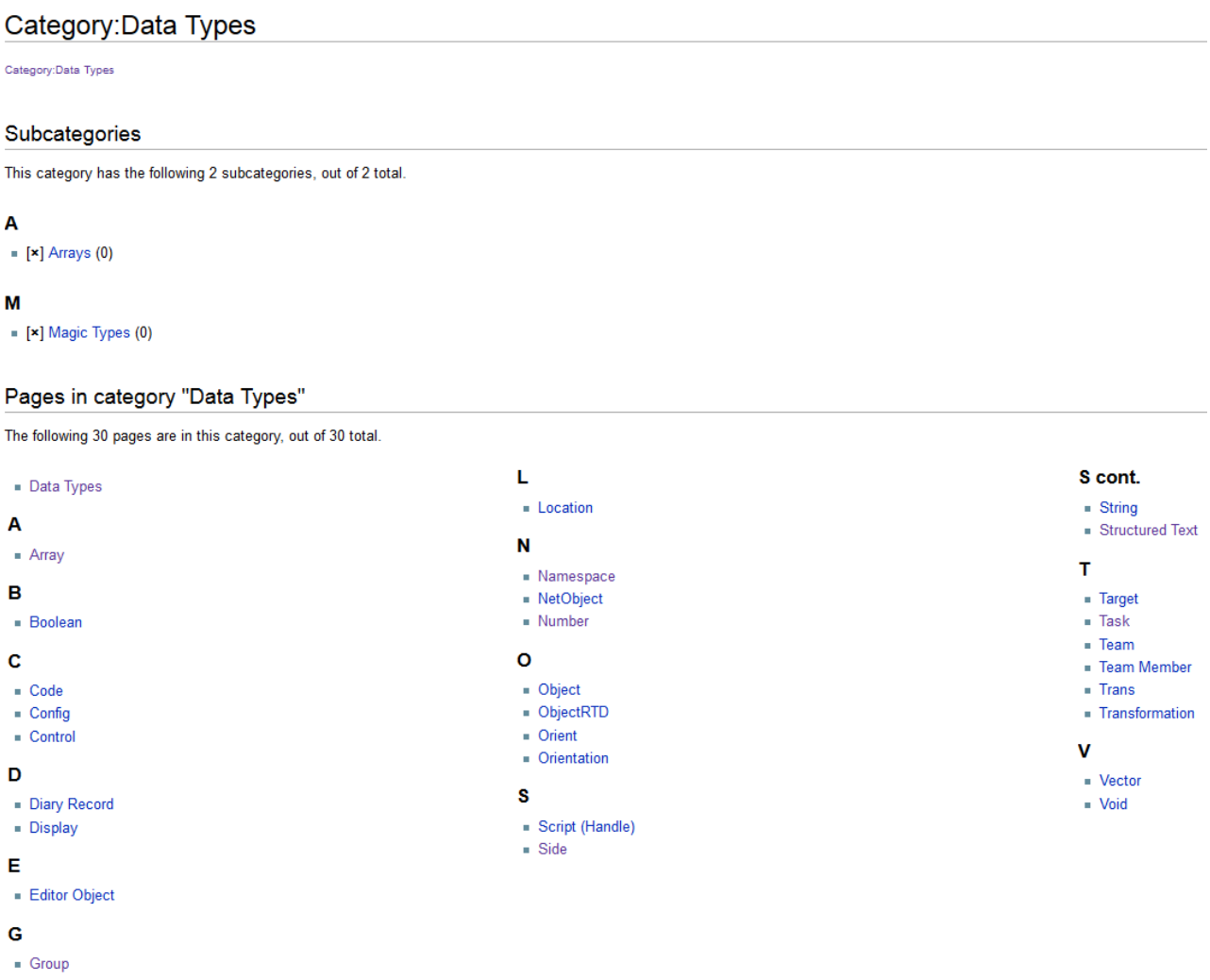
Diese Variante bietet ein Maximum an Benutzerfreundlichkeit, Lesbarkeit und Wartbarkeit, bläht den Code aber auf 5 statt 1 Zeile auf (hier fehlt außerdem noch die -1, was wir uns sparen wollen).

Es gibt noch einige Konventionen zwischen Programmierern, die aber vor allem die Benennung von Variablen, Dateien und dergleichen betreffen. Ehe wir also über diese Dinge sprechen können, wollen wir uns endlich dem Thema Datentypen widmen.

[7] Leider hilft dies nicht, falls RANDOM direkt 0 zurückgibt, aber das ist sehr unwahrscheinlich. Außerdem stehen neben CEIL noch FLOOR und ROUND zur Verfügung, die jeweils auf eine ganze Zahl (ab)runden.

C DATENTYPEN ODER „DAS HANDWERKSZEUG DES PROGRAMMIERERS“

Es ist soweit, wir beginnen mit dem Scripten! Naja…zumindest beginnen wir mit dem, was wir für unsere Scripte brauchen, und das sind Daten oder Werte. Beginnen wir also mit einer Übersicht der Datentypen, die uns ArmA bereitstellt. http://community.bistudio.com/wiki/Category:Data_Types



Category:Data Types

Category:Data Types

Subcategories

This category has the following 2 subcategories, out of 2 total.

A
- [×] Arrays (0)

M
- [×] Magic Types (0)

Pages in category "Data Types"

The following 30 pages are in this category, out of 30 total.

- Data Types

A
- Array

B
- Boolean

C
- Code
- Config
- Control

D
- Diary Record
- Display

E
- Editor Object

G
- Group

L
- Location

N
- Namespace
- NetObject
- Number

O
- Object
- ObjectRTD
- Orient
- Orientation

S
- Script (Handle)
- Side

S cont.
- String
- Structured Text

T
- Target
- Task
- Team
- Team Member
- Trans
- Transformation

V
- Vector
- Void

Was sind Datentypen?

EINE VARIABLE IST EINE REFERENZ AUF ODER EIN CONTAINER FÜR DATEN.

DER DATENTYP EINER VARIABLE LEGT DEREN WERTEBEREICH UND DIE ERLAUBTEN OPERATIONEN FEST.

Das bedeutet, eine Variable kann nur die Werte annehmen, die ihrem Datentyp entsprechen. Und mit ihr können nur Operationen ausgeführt werden, die für diesen Datentyp definiert sind. So wie man in der Mathematik z.B. Vektoren nicht multiplizieren kann, sind auch in ArmA bestimmte Datentypen völlig anders zu behandeln als Zahlen.

Number

NUMBER ist der Standardtyp schlechthin und für lange Zeit wird er unser Hauptdatentyp bleiben. Der Grund: Unter NUMBER verbergen sich **alle** Zahlenformate in ArmA, die in anderen Programmiersprachen noch je nach Wertebereich unterteilt sind und meist Integer und Float heißen. Der Wertebereich für diesen Datentyp reicht von 3.4xE38 bis -3.4xE38. Das bedeutet, ihr könnt getrost jeden Wert damit abdecken. Außerdem beinhaltet der Datentyp in ArmA auch den Subtyp Float, also Fließkommazahlen. Das muss uns aber nicht weiter interessieren, wir haben den Vorteil, dass wir in SQF wirklich jede Zahl als NUMBER ansprechen. Auch Winkelmaße für die trigonometrischen Funktionen SINUS und COSINUS sind NUMBER-Werte.

Wie verwenden wir nun NUMBER? Ganz einfach: Wir brauchen uns über die Deklaration keine Gedanken zu machen! In modernen Programmiersprachen wie JAVA oder VBA muss eine Variable vor der ersten Nutzung (Initialisierung) zunächst deklariert, d.h. bekannt gemacht werden. Normalerweise müsste also zunächst die Variable deklariert (eingeführt) und danach initialisiert (mit Werten belegt) werden:

Listing_8_Java.sqf

```
1   Int a;
2   a = 5;
```

Die erste Zeile gibt bekannt, dass für die Variable a ein Speicherplatz reserviert werden soll, der den Umfang eines Integer hat. Die zweite Zeile ist die eigentliche Wertzuweisung, bei der wir dem Bezeichner „a“ einen Wert 5 (Literal) zuweisen. Das ist das berühmte Schubladenprinzip. Der Vorteil von SQF ist nun, dass wir die erste Zeile niemals brauchen, es gibt keine Deklaration in ArmA (in den meisten Fällen stimmt dies so). Wenn wir eine Variable brauchen, können wir sie direkt mit ihrer Initialisierung einführen und sie ist fortan im Skript oder darüber hinaus bekannt[8].

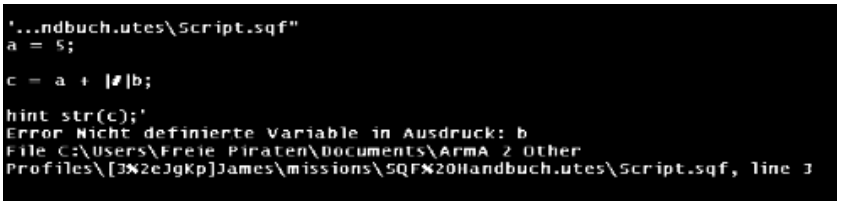
Um jetzt sattelfest in diesem Datentyp zu werden, fehlen uns noch die erlaubten Operationen. Eine Operation ist eine Anweisung, die einen Datentyp verwendet und ihn entweder manipuliert (d.h. verändert) oder aber auswertet (d.h. vergleicht). Um die Begriffe zu verdeutlichen, hier eine Übersicht:

Operationen	Addition	Subtraktion	Multiplikation	Division	Modulo
Operatoren	+	-	*	/	%

Ausdruck	Wert	Kommentar
5+3	8	
5-3	2	
5*3	15	
5/3	1.66667	Anzeige wird beschränkt auf 5 Nachkommastellen
5 % 3 | (5 mod 3)	1	Modulo: Rest einer Division
3 ^ 2	9	Potenzoperator
1/0	0	Produziert einen Laufzeitfehler, wird aber abgefangen.
3*5-2	13	Punkt- vor Strichrechnung
3+5/2	5.5	
3-5-2	-4	Kein guter Stil, da mehrdeutig
(3-5)-2	-4	Besserer Stil
3-(5-2)	0	Eindeutig

Wichtig ist folgende Reihenfolge: Wir können jede Variable erzeugen, die wir brauchen, wenn wir sie mit ihrem Wert einführen. Wir können aber niemals auf eine Variable zugreifen, die vorher nicht eingeführt wurde. Oben haben wir gesagt, dass wir in SQF keine Variable deklarieren müssen, das ist auch richtig. Wir müssen sie aber wenigstens initialisieren, d.h. bei ihrem ersten Auftauchen mit einem Wert versehen. Benutzen wir sie stattdessen in einem Script, gibt es folgende Fehlermeldung:

[8] Dieser Satz bezieht sich auf globale und lokale Variablen. Dieses Thema wollen wir an dieser Stelle noch nicht anschneiden. Es lohnt sich aber, es im Hinterkopf zu behalten.



Auch hier ist die Fehlermeldung dankbarerweise wieder sehr gut nachvollziehbar: „Nicht definierte Variable in Ausdruck: b“. Wir wissen also sogar exakt, welchen Ausdruck er nicht erkannt hat.

Neben diesen arithmetischen Operatoren, die also den Wert einer Variablen verändern, gibt es noch die große Klasse der Vergleichsoperatoren, die ebenfalls sehr wichtig ist:

Vergleichsoperatoren brauchen immer zwei Argumente, zwischen denen sie den Vergleich ausführen. Ganz wichtig ist an dieser Stelle der fundamentale Unterschied zwischen dem einfachen Gleichheitszeichen „=“ und dem doppelten. Das einfache „=“ ist ein Zuweisungsoperator, der einer Variablen den Wert der rechten Seite übergibt. Das doppelte „==“ ist ein Vergleichsoperator, der die beiden Ausdrücke links und rechts vergleicht.

op	*meaning*	true	false
==	*equal*	2 == 2	2 == 3
!=	*not equal*	3 != 2	2 != 2
<	*less than*	2 < 13	2 < 2
<=	*less than or equal*	2 <= 2	3 <= 2
>	*greater than*	13 > 2	2 > 13
>=	*greater than or equal*	3 >= 2	2 >= 3

Mit diesem Wissen können wir bereits in Gedanken Bedingungen und Kontrollstrukturen einführen. Euch fallen sicherlich sofort einige Situationen ein, in denen ihr eure Variablen irgendwie vergleichen und/oder auswerten wollt. Dieses Thema können wir aber erst sinnvoll behandeln, wenn wir den Datentyp BOOLEAN und Kontrollstrukturen in SQF behandeln. Daher bitten wir an dieser Stelle noch um etwas Geduld. Da wir aber gerade Operatoren behandeln, solltet ihr alle wichtigen bereits an dieser Stelle kennenlernen[9].

Viel mehr gibt es zu NUMBER tatsächlich nicht zu sagen.

Boolean

Der Datentyp BOOLEAN kennt nur zwei Werte: **true** (wahr) und **false** (falsch). Daher hat er nur einen Speicherbedarf von einem Bit, weshalb er natürlicherweise oft auch durch die digitalen Zustände 0 und 1 repräsentiert wird. In SQF müssen wir aber immer die Worte TRUE und FALSE verwenden, wenn wir es mit BOOLEAN zu tun haben.

Wozu brauchen wir Wahrheitswerte? Wahrheitswerte haben ihre große Bedeutung, wenn wir Ausdrücke vergleichen und auswerten wollen. Klassische Verwendungen für Boolesche Werte sind Schalter im Sinne von Variablen, die bestimmte Ereignisse steuern sollen oder aber Verzweigungen, die in Abhängigkeit vom Zustand einer Variable etwas tun oder lassen sollen.

Das Besondere bei Booleschen Werten ist ihre verkappte Verwendung. Schauen wir uns folgenden Code an:

Listing_9.sqf

```
1  a = 1;
2  b = 2;
3  Schalter = false;
4  aussage1 = a < b;
5  aussage2 = a > b;
```

[9] Es gibt noch eine dritte Klasse von Operatoren: Die logischen Operatoren. Diesen widmen wir uns beim Typ Boolean.

Die Variablen a und b wurden normal initialisiert, also mit einem Wert versehen. Auch die Variable Schalter wurde initialisiert, da wir aber möchten, dass sie vom Typ BOOLEAN ist, haben wir ihr keine Zahl sondern einen Wahrheitswert zugewiesen. In Zeile 4 und 5 jedoch haben wir keinen direkten Wert angegeben. In Wirklichkeit haben wir zwei verkappte Anweisungen hintereinander ausgeführt.

Eine Zuweisung wird immer von rechts nach links gelesen: Nimm den Wert rechts und schreibe ihn der Variable links zu. Dabei muss der Wert rechts jedoch kein explizit ausgeschriebener Datentyp sein, an jeder Stelle, wo ein Argument erwartet wird, gilt generell, dass auch ein Ausdruck (also eine Berechnung) stehen kann. Das bedeutet in unserem Beispiel, dass die rechte Seite zuerst ausgewertet werden muss (man kann sich generell Klammern um den Ausdruck vorstellen). Die Auswertung a < b liefert aber einen Wahrheitswert zurück, denn der Operator < ist kein arithmetischer, sondern ein boolescher Operator. Da a in der Tat kleiner ist als b, liefert die rechte Seite TRUE und damit haben wir die Variable AUSSAGE1 mit dem Datentyp BOOLEAN angelegt. Demgegenüber legen wir in Zeile 5 eine Variable AUSSAGE2 mit dem Wert FALSE an. Wäre unser Script länger, könnten wir jetzt die Variablen AUSSAGE1 und AUSSAGE2 verwenden, um bestimmte Situationen abzufragen, je nachdem, wofür die Variablen a und b standen.

Zuletzt führen wir noch eine – bereits erwähnte – dritte Klasse von Operatoren ein: die logischen Operatoren:

Diese Operatoren sind aus der Mengenlehre vielleicht dem ein oder anderen bekannt. Im Grunde sind sie „logisch“, führen aber gerade durch die Umgangssprache oft bei Anfängern zu Problemen. Daher noch eine Wahrheitstabelle, die die genaue Verwendung anzeigt:

operations	and	or	not
operators	&&	||	!

a	!a
true	false
false	true

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Besonderes Augenmerk sollte darauf gelegt werden, wann **and** (&&) und wann **or** (||) ausgelöst wird. Übrigens sind die Befehle „and“ und die Abkürzung „&&“ sowie „or“ und „||“ gleichbedeutend. AND löst nur aus, wenn beide Ausdrücke wahr sind. OR löst aus, sobald einer von beiden Ausdrücken wahr ist.

Es gibt noch drei Befehle, die in SQF nicht implementiert sind, die sich aber durch Kombination des eben Gelernten erzeugen lassen:

Befehl	Bedeutung
Xor	((a || b) && !(a && b))
Nor	!(a || b)
Nand	!(a && b)

Wir wollen dies nur der Vollständigkeit halber erwähnen und darauf hinweisen, dass diese Befehle durch eine Wahrheitstabelle nachverfolgt werden können. So kann sich jeder überlegen, dass **nor** gleichbedeutend ist mit „a und b sind beide falsch“. Dies kann wiederrum mit `!(a || b)` oder mit `!a && !b` erreicht werden. Wir sehen: Die Beherrschung der logischen Operatoren ist extrem bedeutsam bei der Abfrage von Wahrheitswerten.

Unser Wissen reicht jetzt für ein kleines Beispielskript:

Listing_10.sqf

```
1  anzahlFeinde = 10;
2  anzahlFreunde = 11;
3  differenz = anzahlFeinde - anzahlFreunde; // Typ Number!
4  fairness = differenz < 0;                 // Typ Boolean!
5  herausforderung = differenz > 0;          // Typ Boolean!
6  hint "Feinde in der Unterzahl:";
7  sleep 1;
8  hint str(fairness);
9  sleep 2;
10 hint "Freunde in der Unterzahl:";
11 sleep 1;
12 hint str(herausforderung)
```

Dieses Script soll einfach nur die eben besprochenen Konzepte rekapitulieren. Es kommen zwei neue Befehle vor, die wir bis dato noch nicht hatten, die sich aber fast intuitiv erschließen: **str**[10] ist ein Befehl, der das Argument in einen STRING umwandelt, dazu kommen wir im nächsten Kapitel; **sleep** ist ein Befehl, der die Ausführung des Scriptes um die angegebenen Sekunden pausiert. Ansonsten solltet ihr alles nachvollziehen können. Wer mag, kann dieses Script selbst einmal im Editor ausprobieren. Wir benutzen die Variable `differenz`, um in Abhängigkeit von der Spieleranzahl (hier Freunde) im Vergleich zur Feindanzahl zu entscheiden, ob die Situation fair ist oder eher ausweglos. Später lernen wir natürlich, statt fiktiver Zahlen die korrekten Verhältnisse auszulesen, momentan soll uns dies aber genügen.

String

STRING ist ein Datentyp für Zeichenketten oder noch allgemeiner: für Text. STRINGs brauchen wir also immer dann, wenn wir Text ausgeben wollen. In seltenen Fällen (wie bei Markern) können wir über STRING auch auf Objekte zugreifen. Manchmal ist auch das Argument eines Befehls vom Datentyp STRING.

Die Zuweisung eines STRINGs erfolgt immer durch den Einschluss des Textes in Anführungszeichen oder alternativ in Hochkommata (‘…‘). Der Text kann alle ASCII-Zeichen enthalten. Bei Umlauten ist darauf zu achten, dass euer Editor auf UTF-8 eingestellt ist, da ansonsten ä, ö, ü und ß nicht korrekt erscheinen. Bereits an dieser Stelle soll darauf hingewiesen werden, dass wir manchmal auch innerhalb von STRINGs Anführungszeichen brauchen, dann müssen diese doppelt ausgeführt werden, da es sonst zu Fehlern kommt, da der Compiler denkt, dass der STRING beim ersten Anführungszeichen im Inneren zu Ende ist.

Eine der wichtigsten Manipulationsmöglichkeiten von STRINGs ist der „+“-Operator. Dabei ist seine Funktionsweise aber ganz anders als der arithmetische Operator des Datentyps NUMBER. Werden zwei STRINGs durch ein + verbunden, dann heißt das schlichtweg, dass die Argumente aneinandergehängt werden. Daher sind folgende zwei Codezeilen vom Ergebnis her identisch:

```
String1 = "Hallo Welt!";
String2 = "Hallo" + " Welt!";
```

Wir müssen lediglich bedenken, dass beim Aneinanderhängen mögliche Leerzeichen eingefügt werden müssen, damit die Teile nicht direkt aneinanderkleben. Der eigentliche Sinn dieser Operation zeigt sich aber erst, wenn wir einen numerischen Wert vom Typ NUMBER ausgeben lassen wollen. Wir hatten bereits oben gesehen, dass dies mit STR möglich ist. Folgender Code würde demnach einen Fehler verursachen:

[10] Leicht zu merken, da STR die Abkürzung für String ist.

```
String = "Es gibt "+5+" Missionsziele";
```

```
'String = "Es gibt " |#|+ 5 + " Missionsziele";'
Error Allgemeiner Fehler in Ausdruck
```

Wie wir sehen, entsteht der Fehler nach dem ersten STRING. Der Fehler „Allgemeiner Fehler in Ausdruck“ ist meist ein Hinweis darauf, dass die grundlegende Syntax verletzt wurde. Wir hatten diesen Fehler bereits im vorherigen Kapitel besprochen, hier taucht er im neuen Gewand auf, da wir jetzt mit dem „+“-Operator einmal einen STRING und zum anderen eine NUMBER zusammenführen wollen. Das ist aber nicht erlaubt.[11] Die Lösung dieses Problems besteht darin, den numerischen Wert ebenfalls in einen String umzuformen. Die erste Eingebung „5“ würde zwar das richtige Ergebnis liefern, machten wir dies aber mit einer Variable – also z.B. „missionsZiele“ – dann würde das Ergebnis unerwünscht ausfallen. Wir müssen vielmehr die Variable MISSIONSZIELE vom Typ NUMBER zum Typ STRING **casten**. Dies geschieht durch den bereits bekannten Befehl STR.

```
String = "Es gibt " + str(missionsZiele) + " Missionsziele";
```

Jetzt stehen links und rechts des Verknüpfungsoperators die gleichen Datentypen. Alles ist gut.[12]

Array

Bisher haben wir uns mit sogenannten primitiven und eindimensionalen Datentypen beschäftigt. Für viele Anwendungen brauchen wir aber einen weiteren, sehr weit verbreiteten Datentyp: Den ARRAY oder das Datenfeld.

Wir erinnern uns an unsere Schubladen-Metapher: Die bisherigen Datentypen waren alle genau eine Kiste, in die wir einen Wert vom jeweiligen Datentyp hineingelegt haben, also eine Zahl, einen Wahrheitswert oder einen Text. Haben wir es aber mit großen Datenmengen (oder noch unbekannten Datenmengen) zu tun, so werden wir allein mit dieser Methode an unsere Grenzen stoßen. Was wir bräuchten, ist ein Regal voller Kisten, ein Schrank voller Schubläden, der schön geordnet vor uns steht und uns jede Schublade anwählen lässt. Klingt nach IKEA, ist aber ein ARRAY.

Ein Datenfeld ist also eine Liste von Elementen, die über einen Index eindeutig angesprochen werden können. Statt Liste könnte man auch Behälter sagen. Zudem ist diese Liste geordnet, d.h. jedes Element steht an einer ganz bestimmten Position in einer Reihenfolge zu den anderen Elementen.

Die letzten zwei Besonderheiten von Datenfeldern in SQF sind die Datentyp-Unabhängigkeit und die dynamische Skalierung. Datentyp-Unabhängigkeit heißt: Ihr braucht nur ein großes Regal und könnt in jede Schublade jeden beliebigen Datentyp ablegen. Ihr braucht euch keine Gedanken darüber machen, dass neben einer NUMBER ein STRING und danach ein BOOLEAN steht, alles möglich. Skalierung bedeutet Anpassung an die aktuelle Größe: Wir müssen dem Datenfeld weder bei seiner Erzeugung mitteilen, wie groß es zu sein hat, noch ist es in seiner Größe beschränkt. Wenn wir neue Elemente hinzufügen, wird das ARRAY vergrößert, wenn wir Elemente entfernen, wird es verkleinert. As easy as this. Damit sind ARRAYs die vielleicht mächtigste weil benutzerfreundlichste Datentyp-Variante.

Wie legen wir jetzt ein Array in SQF an? Ganz einfach:

```
meinArray = [element1, element2, element3,…];
```

Wir bemerken:

EIN ARRAY WIRD DURCH DIE ANGABE VON ELEMENTEN IN ECKIGEN KLAMMERN INITIALISIERT ([…]). ELEMENTE WERDEN DURCH KOMMATA VONEINANDER GETRENNT. DIE ANGABE ERFOLGT EXPLIZIT ODER DURCH EINEN AUSDRUCK.

[11] Ein wichtiger Unterschied zu modernen Programmiersprachen wie Java und C#, die dies erlauben.
[12] Später lernen wir den wesentlich mächtigeren Befehl FORMAT kennen, den wir hier aber nicht einführen wollen. Für Interessierte: Kapitel I. 9

Wir werden später sehr oft auf ARRAYs stoßen, auch dort, wo wir sie nicht direkt erkennen, weil die eckigen Klammern eine Art Einheit signalisieren, die bei manchen Befehlen nicht sofort an ein ARRAY denken lässt. Daher lohnt es sich, sich klarzumachen, dass eckige Klammern immer ein ARRAY erzeugen.

Wir erinnern uns an die obige Bemerkung, dass jedes Argument/Element auch durch einen arithmetischen oder logischen Ausdruck ersetzt werden kann. Statt also konkrete Werte anzugeben, können wir auch innerhalb des Datenfeldes diese Werte erst berechnen oder auslesen lassen:

```
meinArray = [3+5, a < b, str(missionsZiele)];
```

Dieses ARRAY besteht aus genau 3 Elementen. Der Compiler weiß, dass er bei `3+5` zuerst die Summe berechnen muss, ehe er das Ergebnis als 1. Element übernehmen kann. Ebenso ist ihm bewusst, dass das 2. Element erst bekannt ist, nachdem er den Ausdruck `a < b` ausgewertet hat. Wer unsicher ist, darf aber gerne diese Ausdrücke in runde Klammern einschließen. Wir haben zunächst ein Element vom Typ NUMBER, danach einen Vergleichsoperator, der einen BOOLEAN liefert und schlussendlich einen STRING, den der Befehl STR erzeugt. Wir sind also wirklich frei, was die Erzeugung von Datenfeldern angeht.

An dieser Stelle möchten wir auch kurz erwähnen, dass multidimensionale ARRAYs möglich sind. Das heißt nichts anderes, als dass wir als Elemente innerhalb eines Datenfeldes wieder Datenfelder benutzen dürfen. Sobald wir ein Element nämlich in eckige Klammern einschließen, haben wir ein neues Datenfeld erzeugt.

```
meinArray = [element1, element2, [arrayelement1, arrayelement2]];
```

Hier sind der Kreativität keine Grenzen gesetzt, selbst dreidimensionale ARRAYs und mehr sind möglich. Wir werden später darauf eingehen, wie wir jetzt auf diese Elemente zugreifen können.

Nun stellt sich natürlich die Frage, welche Operatoren für Datenfelder erlaubt sind. Und auch hier stoßen wir zunächst auf den „+“-Operator. Mit unserer bisherigen Erfahrung bei NUMBER und STRING können wir erahnen, was dieser tut: Er heftet zwei ARRAYs aneinander. Dabei verlängert „+“ jeweils das linksstehende ARRAY um die Elemente des rechtsstehenden. Dies geht natürlich wie immer nur, wenn die Argumente beide vom Datentyp ARRAY sind. Der vielleicht anfängliche Impuls

```
meinArray = [element1] + element2;
```

wird scheitern, da `element2` von irgendeinem Datentyp ist, aber nicht vom Typ Array. Korrekt heißt es:

```
meinArray = [element1] + [element2];
```

Das Ergebnis ist ein ARRAY der Form `[element1, element2]`. Würden wir die Reihenfolge herumdrehen, wäre `element2` natürlich vorn an erster Stelle, denn wie erwähnt ist der „+“-Operator linksassoziativ, was heißt, dass er auf die Reihenfolge achtet.

Möchten wir ein Element entfernen, steht uns prinzipiell der „-“-Operator zur Verfügung. Sein Nachteil besteht darin, dass er eher wie der Mengenoperator „ohne“ funktioniert. Konkret: „-“ löscht aus einem ARRAY **alle** Elemente, die in beiden ARRAYs vorkommen. Also:

Listing_11.sqf

```
1  Array1 = [1,2,3,4,3,2,1];
2  Array2 = [1,2,3];
3  Array3 = Array1 - Array2
```

`Array3` besteht nach Ausführung nicht etwa aus `[1,2,3,4]`, sondern nur noch aus `[4]`. Eine bessere Möglichkeit, gezielt einzelne Werte zu manipulieren, werden wir weiter unten kennenlernen.[13]

[13] Sollte sich ein Leser an dieser Stelle wirklich fragen, ob er verschachtelte, also mehrdimensionale Arrays voneinander abziehen kann, so lautet die Antwort nein. Workaround kann im BIS Wiki nachgelesen werden: Das entsprechende Element muss zunächst in einen nicht-Arraytyp umgewandelt werden. Dies geht mit dem SET-Befehl.

Wir sind jetzt in der Lage, ARRAYs anzulegen und zu manipulieren, sie also zu erweitern oder zu verkürzen. Besprechen wir noch einen anderen Befehl und schauen uns dann an, wie wir einzelne Elemente auslesen bzw. auf diese zugreifen können.

Der „+"-Operator kann noch in der Funktion des Kopierens gebraucht werden. Dies geschieht immer dann, wenn links vom „+" kein Argument steht. Konkret also:

```
Array2 = + Array1;
```

Diese Zeile besagt, dass der Inhalt von `Array1` in `Array2` kopiert wird. Beide Arrays sind danach aber unabhängig voneinander, können also auch getrennt manipuliert werden. Warum ist das wichtig? Nun, der Code:

```
Array2 = Array1;
```

macht augenscheinlich dasselbe. Dies ist aber nicht der Fall, denn bei ARRAYs haben wir es mit Referenzvariablen zu tun. Beim Datentyp NUMBER war das Schubladenprinzip so, dass wir den Wert rechts vom „="-Operator mit dem Bezeichner links vom „="-Operator fest verbunden hatten. `a = 5` heißt, dass `a` den Wert `5` hat, wir können `a` manipulieren und verändert damit auch immer die `5`. Bei ARRAYs ist das etwas anders, man spricht von Pointern oder eben Referenzen: Der Bezeichner – also der Name der Variable – zeigt nicht direkt auf den Wert bzw. hat nicht selbst den Wert des Datenfeldes, sondern er zeigt nur auf dessen Speicherplatz. Das bedeutet, wir können verschiedene Bezeichner benutzen, die alle auf dasselbe ARRAY zeigen. Verändern wir dann das ARRAY über einen Bezeichner, so zeigen alle anderen Bezeichner ebenfalls auf das veränderte ARRAY. Ein wichtiger Unterschied!

Um dazu ein Beispiel machen zu können, brauchen wir aber erst den Befehl, der es uns ermöglicht, ein Datenfeld an einer beliebigen Stelle zu verändern. Vielleicht denkt ihr jetzt an den „+"-Operator, aber dieser erzeugt in Wirklichkeit ein neues ARRAY! Daher bezieht sich der obige Absatz auch nicht auf diesen Operator, denn dort tritt das Problem nicht auf. Wir werden gleich nachvollziehen, was damit gemeint ist, aber zunächst der Befehl **set**:

Description

Description:	Format of element is [index, value]. Changes an element of given array. If element does not exist, resize index+1 is called to create it.

Syntax

Syntax:	array **set** [index, value]
Parameters:	array: Array [index, value]: Array index: Number value: Anything
Return Value:	Nothing

Examples

Example 1:

```
_arrayOne set [0, "Hello"]
```

Dieser Befehl erlaubt uns also (Description lesen!), ein beliebiges Element innerhalb eines ARRAYs zu verändern. Und er ist sogar noch mächtiger! Der letzte Satz verrät, dass selbst die an sich fehlerhafte Angabe einer Position ausgeführt wird, indem das ARRAY um die fehlenden Einträge ergänzt wird. Das Beispiel verdeutlicht die Benutzung: Vor dem Operator muss eine Variable vom Typ ARRAY stehen, danach folgt ebenfalls ein ARRAY, das aus zwei Einträgen besteht: Der Position und dem neuen Wert. Mit anderen Worten: Wir verändern das 1. Element von _ARRAYONE[14] von seinem alten Wert zu „Hello".

[14] Warum wir vor der Variable einen Unterstrich _ haben, kommt im Kapitel I. 4.C

Warum das 1. Element? Ein wichtiger Hinweis fehlt uns noch:

DER INDEX VON DATENFELDERN BEGINNT IMMER BEI 0!

Das bedeutet, dass das 1. Element den Index 0 erhält, das zweite den Index 1 usw. Demnach ist klar, dass wir eine 0 als Index angeben müssen, um das erste Element zu verändern.

Jetzt können wir auch zu unserem Problem mit dem Kopieren von ARRAYS zurückkehren. Wir hatten gesagt, dass der „="-Operator keine neue Variable anlegt, sondern quasi beide Variablen auf dasselbe ARRAY zeigen lässt, weil ARRAYS nur Referenzvariablen sind. Dazu ein Beispiel:

Listing_12.sqf

```
1  Array1 = [1,2];
2  Array2 = Array1; // Jetzt zeigen beide Bezeichner auf denselben Speicherort!
3  Array1 set [2,3] // Wir verändern Array1 (wirklich?)
```

Zeile 1 und 2 sollten euch soweit bekannt sein. Wären wir bei NUMBER, so wäre `Array2` wirklich eine eigenständige Variable mit eigenem Wert. Wir sind aber bei Datenfeldern, und Zeile 2 bewirkt nichts anderes, als dass `Array1` seine Referenz an `Array2` kopiert, d.h. beide Variablen zeigen jetzt auf denselben Speicherort! In Zeile 3 benutzen wir den neuen Befehl, um `Array1` zu verändern. Wir geben als zu änderndes Element den Index 2 und damit das 3. Element an, das noch nicht existiert. Daher wird `Array1` um eine Position erweitert und das 3. Element erhält den Wert `3`. Ergebnis (ACE-Debug-Konsole benutzen!) lautet: `Array1 = [1,2,3]`. Das spannende an der Geschichte ist, was mit `Array2` passiert. Wäre es unabhängig, so würde sein Wert nach Zeile 2 bei `[1,2]` bleiben. Verfolgt man aber dessen Wert, so wird man sehen, dass `Array2` sich in dem Moment mit `Array1` ändert, wo `Array1` per SET angepasst wird. Das heißt, auch `Array2` lautet jetzt `[1,2,3]`! `Array1` und `Array2` sind also zwei verschiedene Bezeichner, die auf ein und dieselbe Referenz verweisen. Daher gibt es den Copy-Operator:

Listing_13.sqf

```
1  Array1 = [1,2];
2  Array2 = + Array1; // unärer Operator
3  Array1 set [2,3]
```

In diesem Beispiel hat sich nur eine Winzigkeit geändert: Statt dem Zuweisungsoperator „=" haben wir den unären Operator „+" verwendet. Unär bedeutet, er erwartet nur ein Argument, im Vergleich zu binär, was zwei Argumente bedeuten würde, also vor und nach dem Operator. Da der „+"-Operator eine Kopierung des Inhaltes von `Array1` nach `Array2` bewirkt, sind diese danach wirklich unabhängig voneinander. `Array2` zeigt also weiter auf ein ARRAY mit dem Inhalt `[1,2]`, auch wenn `Array1` auf ein neues ARRAY mit dem Inhalt `[1,2,3]` zeigt. Wir haben die Bezeichner also entkoppelt.

Als letztes bleibt uns nur noch zu erwähnen, wie wir Elemente aus ARRAYS auslesen können. Wir wissen, wie wir sie anlegen, wie wir sie erweitern („+") oder verkürzen („-") und wie wir ein Element ändern können (SET). Das Auslesen von Elementen geht über den Befehl **select**.

Description

Description:	Selects an index element of an array or config object. Index 0 denotes the first element, 1 the second, etc. If index has decimal places it gets rounded down for fractions less than .5, otherwise it gets rounded up.

Syntax

Syntax:	array **select** index
Parameters:	array : Array index: Number
Return Value:	Any Value

Select ist also ebenfalls ein binärer Operator, wie wir auch erwarten würden. Zuerst muss natürlich bekannt gegeben werden, aus welchem ARRAY wir etwas selektieren wollen. Anschließend müssen wir das Element bekannt geben, das wir auswählen wollen. Dies geschieht wieder über den bereits bekannten Index, beginnend bei 0. Interessant ist der Hinweis, dass es sich beim Index um keine natürliche Zahl handeln muss, der Index wird nach den Rundungsregeln automatisch auf die ganze Zahl gerundet.

Damit sind wir in der Lage, Elemente sowohl aus eindimensionalen als auch aus mehrdimensionalen Arrays auszulesen, wie wir gleich ausprobieren wollen:

Listing_14.sqf

```
1  Array1 = [1, "Zwei", 2+1]; // Array mit 3 Elementen
2  Array1 select 0; // liefert 1
3  Array1 select 1; // liefert "Zwei", da vom Datentyp String
4  Array1 select 2; // liefert 3, wurde also berechnet
5  Array2 = [[1,2],[3,4]]; // 2D-Array
6  Array2 select 0; // liefert [1,2], da vom Datentyp Array
7  Array2 select 1  // liefert [3,4], da vom Datentyp Array
```

Wir haben zwei ARRAYs angelegt und jeweils alle Elemente ausgelesen. Die Kommentare sollten an dieser Stelle als Erklärung genügen. Die Übungen am Ende dieses Abschnitts werden darauf eingehen, wie man bei `Array2` auch direkt auf die Elemente innerhalb des inneren ARRAYs zugreifen kann.

Wenn wir später Schleifen behandeln werden, solltet ihr euch schon jetzt gut merken, dass der Index von ARRAYs immer bei 0 anfängt. Das ist sowohl beim Zugriff als auch beim Durchlaufen aller Elemente wichtig. Denn wenn wir später einmal über alle Elemente sozusagen iterieren, d.h. jedes Element auslesen wollen, so müssen wir als Länge des ARRAYs immer eins weniger als die Anzahl der Elemente angeben (n-1)[15].

Weitere Datentypen, die man kennen sollte

Machen wir es kurz: Ja, es gibt noch einige Datentypen (s.S. 27), aber diese braucht ihr nur bei einigen weniger häufig benutzten Befehlen oder sie ergeben sich als eine Kombination aus den bisher genannten – oder sie sind einfach kaum von Bedeutung.

Wir werden kurz ein bis zwei Sätze zu den anderen Datentypen sagen, wo wir eine Bemerkung für sinnvoll halten. Wir gehen dabei chronologisch durch die Übersicht von Seite 27.

Code ist streng genommen kein eigener Datentyp, dennoch kann man ihn formal als ein Argument eines Befehls wie des FOREACH-Befehls auffassen. Demnach muss dann als Argument eine lauffähige SQF-

[15] An dieser Stelle sei der Befehl COUNT erwähnt, der euch bei vielen Zählproblemen eine gute Hilfe sein kann! Außerdem kann man schon einmal erwähnen, dass wir für einen Durchlauf aller Elemente eines ARRAYs auch FOREACH nutzen können und damit gar keine Information über die Länge benötigen.

Anweisung übergeben werden. CODE wird meist in geschweifte Klammern gefasst und von der Engine precompiled, das heißt bereits vorübersetzt. Dies wird uns bei Funktionen wiederbegegnen.

Control begegnen wir bei der Programmierung von eigenen User-Interfaces, auch Dialogen genannt. Dabei handelt es sich um graphische Menüs, die wir im Spiel aufrufen können, um z.B. die Handhabung komplexere Scripte oder Benutzereingaben zu ermöglichen. Wir werden ebenfalls viel später zu diesem Thema kommen. Eng verwandt mit dem CONTROL ist das DISPLAY.

Display ist, wenn man so möchte, der übergeordnete Datentyp zu CONTROL. DISPLAY kann CONTROLS und Dialogs enthalten und ist quasi der Bildschirm oder eben das Display, auf dem das Spiel gerade stattfindet. Wird eigentlich auch nur für graphische Dialoge gebraucht.

Group ist relativ wichtig und häufig, wenn ihr euch mit Respawn- oder Wegpunkt-Befehlen befasst, bzw. allgemein das verhalten einer Gruppe beeinflussen wollt. GROUP repräsentiert dabei als eigenständiger Datentyp die Gruppe, zu der alle Einheiten gehören. GROUP kann also tatsächlich aus einer einzigen Einheit oder eben aus einer ganzen Gruppe von vielen Einheiten bestehen.

Namespace ist quasi so etwas wie der Gültigkeitsbereich für Variablen. Es gibt verschiedene NAMESPACE-Arten, einige mit sehr interessanten Möglichkeiten, wie dem PROFILENAMESPACE. Dies würde aber diese Einführung sprengen, wir verweisen auf spätere Kapitel. Eng verwandte Befehle für diesen Datentyp sind SETVARIABLE und GETVARIABLE. Wir kommen weiter unten darauf zurück.

Object ist die Oberkategorie für alle Objekte im Spiel, die quasi in 3D manipuliert werden können. Dazu gehören Einheiten, Gebäude, Vegetation, Krater, also generell alles, was im Editor platziert wird oder eine ID hat. Erfordert ein Befehl also als Argument ein OBJECT, so könnt ihr unterschiedliche „Dinge“ wie den Spieler (PLAYER) oder ein Gebäude angeben.

Side ist ein sehr praktischer Datentyp, den wir vor allem in Zusammenhang mit Kontrollstrukturen und vordefinierten Einheiten-ARRAYS wie ALLUNITS gebrauchen werden. SIDE ist die Seitenzugehörigkeit eines Objects und kann demgemäß die Werte West, East, Civilian, Resistance, sideLogic, Friendly, Enemy oder Unknown annehmen.

Task ist der Datentyp für Missionsziele und wird auch nur in diesem Zusammenhang gebraucht. Missionsziele können angelegt, verändert, gelöscht oder aktualisiert werden. Außerdem kann ihr Status über die entsprechenden Befehle aktualisiert werden.

Abschließende Feinheiten 

Wir haben eingangs bereits darauf hingewiesen, dass es trotz der nicht notwendigen Deklaration einer Variablen sehr wohl nötig ist, jede Variable, die man benutzen möchte, zu initialisieren. Ein Operator, der auf eine unbekannte Variable stößt, wird eventuell einen Fehler verursachen, viel öfter wird er aber wortlos seinen Dienst einstellen und das ganze Script kann so abgebrochen werden. Auch wenn wir die Struktur noch nicht hatten, hier ein Beispiel, wovon wir reden:

```
if (player == s1) then {anweisung};
```
[16]

Unabhängig davon, was wir mit dieser Zeile genau bezwecken wollen, ist folgende Feststellung fundamental: Wenn dem Script die Variable s1 nicht bekannt ist, wird das Script nicht weiter ausgeführt! Dies kann vor allem in Auslösern zu Problemen führen, die als Auslösebedingung an eine bestimmte Einheit gebunden sind. Im Editor ist die KI standardmäßig eingeschaltet, d.h. jede Einheit ist dem Script auch bekannt. Im MP kann es dann aber vorkommen, dass nicht jede Einheit von einem Spieler besetzt ist. Betrifft

[16] Siehe Kapitel I. 4.B: Hinter `anweisung` muss kein Semikolon, sofern es die letzte/einzige Anweisung ist, sonst `{… ; …}`.

dies gerade die Einheit, die aber in einem Script als Variable eine Rolle spielt, wird diese Script nicht funktionieren.[17]

Wir möchten euch also dafür sensibilisieren, gut darüber nachzudenken, welche Variable ihr an welcher Stelle benutzt und ob diese möglicherweise nicht bekannt ist.

Wie kann man nun aber solche Probleme umgehen? Wie kann man verhindern, dass unbekannte, nicht definierte Variablen benutzt werden? Dafür müsste man natürlich eine Möglichkeit haben, den Status einer Variable zu überprüfen. Genau dafür steht uns **isNil** zur Verfügung:

Description

Description:	Tests whether the variable defined by the String argument is undefined, or whether an expression result passed as Code is undefined. expression result undefined (i.e. the expression result is Void), and false in all other cases.

Syntax

Syntax:	Boolean = **isnil** variable
Parameters:	variable: String or Code
Return Value:	Boolean

Examples

Example 1:

```
if (isnil ("_pokus")) then {_pokus=0;}
```

Wie wir sehen können, testet ISNIL eine Variable oder einen Ausdruck darauf, ob er definiert, d.h. mit irgendeinem Wert belegt ist. Da der Rückgabewert ein BOOLEAN ist, gibt ISNIL TRUE zurück, wenn die Variable nicht definiert ist und FALSE, wenn sie definiert ist. Obwohl das Beispiel auf einen IF-Befehl zurückgreift, den wir noch nicht hatten, kann die Bedeutung vielleicht trotzdem erahnt werden. Wir bitten euch einfach, ISNIL als Möglichkeit im Kopf zu behalten, eine Variable zu überprüfen.[18]

Nun wissen wir, wie wir Variablen erzeugen und überprüfen können. Was uns fehlt, ist eine Möglichkeit, sie wieder zu löschen bzw. zur Löschung freizugeben. Dies geschieht über eine Wertzuweisung mit dem Wert **nil**. Wenn wir also eine Variable nicht mehr benötigen, können wir ihr den Wert NIL übergeben und sie wird von der Engine gelöscht. Bitte überschreibt daher niemals NIL selbst! Also kein `nil = irgendetwas`.

Abschließend möchten wir noch für einen guten Programmierstil werben. Wir haben bereits zu Beginn dieses Kapitels einige Konventionen kennengelernt: Kommentare schreiben, Anweisungen immer in eine neue Zeile schreiben, Blöcke nutzen usw. Wir möchten jetzt noch zwei weitere Konventionen hinzufügen.

Die erste besagt, dass wir verschachtelte Blöcke einrücken. Das bedeutet, dass wir generell einen Tabstopp vor eine Anweisung schreiben, wenn diese im Inneren einer anderen Einweisung steht. Dies erhöht die Übersichtlichkeit und Lesbarkeit enorm und hilft schnell, Abhängigkeiten sichtbar und nachvollziehbar zu machen.

Die zweite betrifft die Benennung von Bezeichnern alias Variablen. Prinzipiell können Variablen in SQF beliebig benannt, Groß- und Kleinschreibung beliebig kombiniert werden. Auch wenn SQF **nicht** auf Groß- und Kleinschreibung achtet, so hat sich doch folgende Konvention durchgesetzt:

[17] Eigene Versuche des Autors deuten darauf hin, dass dieses Problem nur bei Kontrollstrukturen besteht. Ist ein Element innerhalb eines Arrays nicht bekannt, wird das Skript trotzdem weiter ausgeführt – mit einer Fehlermeldung.
[18] Achtet bei diesem Befehl unbedingt darauf, die zu überprüfende Variable als String anzugeben.

Variablen werden nach ihrem Inhalt oder ihrer Verwendung benannt. Der Name besteht aus Kleinbuchstaben, solange ein einzelner Begriff verwendet wird. Bei zusammengesetzten Begriffen wird der zweite Begriff mit einem Großbuchstaben versehen. Konstanten bestehen nur aus Großbuchstaben.[19]

Dies erscheint auf den ersten Blick etwas umständlich oder aufwändig, führt aber sehr schnell zu einer intuitiven Vergabe von Namen, die zudem sehr gut von anderen Programmierern nachvollzogen werden kann. Eine Variable sollte also immer aussagekräftig benannt werden (`zähler` für eine Variable, die in einer Schleife zählt usw.).

Wir werden uns ab jetzt an diese Konventionen halten und unsere Scripte dementsprechend hier abbilden.

Namespaces

Jetzt kommen wir auf ein sehr wichtiges Thema zu sprechen: Dem Gültigkeitsbereich von Variablen. Bisher haben wir uns darüber keinerlei Gedanken gemacht – wir haben ja auch noch keine Scripte bisher geschrieben. Sobald wir aber tiefer in die Materie einsteigen, müssen wir uns Gedanken über den Gültigkeitsbereich von Variablen machen. Es wäre z.B. ja völlig überflüssig, eine Zählervariable, die wir nur innerhalb einer Schleife brauchen, an alle Spieler in einer Multiplayer-Sitzung zu senden. Ebenso wäre es aber auch nicht sinnvoll, eine wichtige Variable, die wir in mehreren Scripts brauchen, jedes Mal neu zu berechnen oder umständlich zu übergeben. Mit anderen Worten: Es muss globale und lokale Variablen geben.

Es gibt drei verschiedene Namespaces: Lokal, Global und Public.

Lokale Variablen sind immer nur im jeweiligen Script sichtbar. Ein Script außerhalb kann nicht auf diese Variable zugreifen und die Variable selbst wird mit dem Ende des Scripts, in dem sie definiert wurde, auch wieder gelöscht.

Globale Variablen sind über Scripte hinweg während der ganzen Mission aktiv und bekannt. Solange wir nicht im Teil II – Lokalität sind, muss uns zunächst nicht interessieren, wie es dabei mit anderen Spielern aussieht. Daher reicht uns die Faustformel: Soll eine Variable in mehreren Scripten zur Verfügung stehen, sollten wir sie global anlegen.

Öffentliche Variablen sind spezielle globale Vertreter, die zusätzlich an alle Spieler einer Mehrspieler-Partie übertragen werden. Damit gehören sie in Teil II und werden hier nicht näher behandelt.

Lokale und globale Variablen sind nun ganz einfach zu unterscheiden bzw. anzulegen:

Listing_15.sqf

```
1  Array1 = [element1, element2];  // globale Variable, bekannt für den Client
2  _localVar = count Array1;       // lokale Variable, bekannt nur für das Skript
```

Dieser Zweizeiler legt zwei Variablen an. Die erste ist ein bereits bekanntes ARRAY und global, denn sie steht ohne einen Präfix am Zeilenanfang. Die zweite (`_localVar`) ist lokal, denn sie beginnt mit einem Unterstrich (_). Wenn wir also nichts tun, ist eine Variable immer global, lokale Variablen hingegen müssen wir quasi extra bekanntgeben (ein wichtiger Unterschied zu modernen Programmiersprachen wie Java).

Damit steht uns durch den **count**-Befehl[20] die Länge des ARRAYs zur Verfügung, diese Variable brauchen wir aber nur innerhalb des Scriptes und definieren sie daher lokal. Sie wird am Ende gelöscht. Das ARRAY selber wollen wir aber auch in anderen Skripten benutzen, daher haben wir es ohne ein Präfix direkt `Array1` genannt. Schrieben wir jetzt also ein zweites Script, so könnten wir in diesem `Array1` direkt als Variable verwenden, mit den oben genannten Einträgen. `_localVar` hingegen wäre dort nicht bekannt

[19] Die Forderung, Variablen nur noch englisch zu benennen, wollen wir hier nicht vertreten.
[20] Wir hatten diesen Befehl bereits früher erwähnt und führen ihn jetzt offiziell ein. Da er aber über einige mächtige Eigenschaften verfügt, wollen wir auch nicht zu sehr in die Tiefe gehen und verweisen auf später.

und könnte auch mit gleichem Namen wieder angelegt werden. Initialisieren wir aber `Array1` erneut, so würden wir die globale Variable ändern, also auch eventuell noch laufende Scripte, die bereits mit `Array1` arbeiten! Hier ist Vorsicht geboten.

Es gibt noch eine Besonderheit im Zusammenhang mit Namespaces, die erwähnt werden muss, sobald man sich tiefer in die Materie begibt und unnötige Fehler vermeiden will: lokale Variablen innerhalb von Kontroll-Struktur-Blöcken gelten nur innerhalb dieser Kontroll-Struktur! Wir greifen an dieser Stelle bereits wieder auf das nächste Kapitel vor, weil es leider nicht ohne Kontrollstruktur geht:

Listing_16.sqf

```
1  if (alive player) then {_living=true};
2  hint str(_living)
```

In der ersten Zeile wird der Zustand des Spielers abgefragt. Dazu ist der Befehl **player** notwendig. Ist der Spieler am Leben, wird eine lokale Variable auf wahr gesetzt. Dies soll anschließend ausgegeben werden. Problem: `_living` ist außerhalb der Kontrollstruktur nicht bekannt und noch viel gemeiner: die Ausgabe mittels HINT-Befehl führt noch **nicht** einmal zu einem Scriptfehler. Dies wäre nur der Fall, wenn die Variable nicht vorhanden ist. Sie existiert aber, nur nicht dort, wo wir auf sie zugreifen wollen. Wem dies passiert, der kann sich sehr lange an der Fehlersuche aufhängen, wenn er nicht um diese Tatsache weiß. Also entweder alle Befehle in den Kontrollstruktur-Block verlegen oder die lokale Variable außerhalb der Kontrollstruktur initialisieren. Dann steht sie auch innerhalb der Verzweigung zur Verfügung, aber eine Änderung im Innern ist nicht länger nur im Innern gültig, sondern kann auch außerhalb abgefragt werden.

Übungen

[1] Eine Übung für arithmetische Operatoren: Wie könntet ihr überprüfen, ob ein gegebenes Jahr ein Schaltjahr ist, wenn ihr wisst, dass ein Schaltjahr durch 4 teilbar sein muss?

[2] Wie könntet ihr überprüfen, ob der Spieler (PLAYER) eine bestimmte Einheit ist? Wie könntet ihr ausschließen, dass er eine bestimmte ist? Wie könntet ihr eine dieser Bedingungen mit der Zusatzbedingung verbinden, dass der Spieler eine Westeinheit ist (notwendiger Befehl: **side**)?[21]

[3] Wie versprochen wollten wir noch nachreichen, wie ihr auf zweidimensionale ARRAYs zugreifen könnt. Im Kapitel Array haben wir mehrdimensionale ARRAYs vorgestellt und dazu den Befehl SELECT. Die Aufgabe besteht nun darin, direkt auf ein Element im Inneren eines verschachtelten ARRAYs zuzugreifen.

[4] Erzeugt aus dem gegebenen `Array1` das ebenfalls angegebene `Array2`:

```
Array1 = ["Text",2,true];
Array2 = [1,false,true,4];
```

[5] Wie könnt ihr das letzte Element eines ARRAYs auslesen, ohne dessen Länge kennen zu müssen?

[21] Für alle Übungen gilt generell, dass ihr benötige Variablen nach Belieben definieren könnt, denkt aber an die Gültigkeit.

I. 5. KONTROLLSTRUKTUREN

Jetzt wird es spannend! Ehrlich. Wirklich. Also im Ernst jetzt. Bisher haben wir uns durch die eher trockenen und doch so notwendigen Grundlagen der Scriptsprache durchgearbeitet, um die vielen Befehle in ArmA korrekt und ohne große Fehler anwenden zu können. Wir kennen sehr viele Datentypen und wissen, wie wir uns in die exotischeren einlesen können und worauf wir achten müssen. Wir können bereits einige Befehle sinnvoll benutzen und haben auch gelernt, wie wir Variablen ressourcenschonend anlegen.

Was jedoch aus einem Script erst ein richtiges Script und letztendlich ein sinnvolles Stück Funktionalität macht, das sind Kontrollstrukturen. Erst sie können überhaupt all die Datentypen, die wir bis jetzt gelernt haben, sinnvoll auswerten. Ohne Kontrollstrukturen wären wir auf eine lineare Abarbeitung von Befehlen eingeschränkt und könnten niemals Bedingungen stellen oder Variablen auswerten. Kurz gesagt: SQF wäre ziemlich langweilig und kaum der Rede wert.

Kontrollstrukturen regeln – einfach ausgedrückt – den Ablauf eures Scriptes, also in welcher Reihenfolge Befehle abgearbeitet werden. Ohne Kontrollstrukturen müssten wir immer jeden Befehl abarbeiten, wir könnten keinen überspringen oder mehrfach ausführen. Kontrollstrukturen helfen uns dabei, unsere Scripte zu strukturieren und mit Verzweigungen sowie Schleifen zu versehen. Daher werden Kontrollstrukturen auch in diese zwei Klassen eingeteilt, die wir uns jetzt in Ruhe ansehen wollen. Ich empfehle einen Kaffee…oder ein Stück Kuchen.

A VERZWEIGUNGEN

Die einfachste Kontrollstruktur ist die Verzweigung. Eine Verzweigung teilt das Ablaufschema eines Scriptes, mit anderen Worten: Der Code wird nicht mehr einfach linear Zeile für Zeile abgearbeitet, sondern ein Teil des Codes kann übersprungen werden, wenn die genannte Bedingung nicht erfüllt ist. Somit sind wir erstmals in der Lage, bestimmten Code nur dann auszuführen, wenn eine Bedingung erfüllt ist.

Die Verzweigung als if-else-Bedingung

Die klassische Struktur für diese Verzweigung ist in fast allen Programmiersprachen die **if**-Anweisung. Die Struktur ist tatsächlich selbsterklärend, die englischen Begriffe bezeichnen exakt, was passiert. Dennoch wollen wir uns wieder zunächst den Steckbrief des Befehls anschauen:

Description

Description: The standard if, then, else construct available in many languages. This syntax however has alternate forms in the manner of an Array.

- if (condition) then {code} else {code}
- if (condition) then [{code},{code}]

Result of the Code executed is returned as the result to this command (which may be Nothing}.

Syntax

Syntax: Anything = **if** (condition) **then** {Code} **else** {Code}

Anything = **if** (condition) **then** **[** {ThenCode} , {ElseCode} **]**

Parameters: condition: Boolean expression. If it evaluates as 'true', the 'then' clause is executed. Otherwise, the 'else' clause (if present) is executed.

Return Value: If Type - Predicate which will execute 1st or 2nd option when used. This predicate is used in then or exitWith commands.

Wir sehen, dass es prinzipiell zwei verschiedene Syntax-Varianten gibt: Die erste besteht aus der Bedingung, gefolgt von der THEN-Anweisung und zum Schluss der ELSE-Block für eine nicht erfüllte Bedingung. Alternativ kann man sowohl den THEN- als auch den ELSE-Block in eckige Klammern zusammenfassen. Wir raten von dieser Methode aber aus mehreren Gründen dringend ab. Allein für die Lesbarkeit und Fehlersuche sollte man bei der ersten Variante bleiben.

Prinzipiell ist dies aber die vollständige Syntax, die nicht immer notwendig ist. Das soll heißen, dass ihr durchaus auch nur den THEN-Block beschreiben müsst und den ELSE-Block weglassen könnt. Wenn ihr

ausschließlich Code im Falle einer zutreffenden Bedingung ausführen wollt, auf den ELSE-Teil aber verzichten könnt, ist das nicht verboten.

Damit könnten wir ein einfaches Standard-Beispiel betrachten:

Listing_17.sqf

```
1  if (1 > 0) then
2  { "true" }
3  else
4  { "false"};
```

Wir prüfen zunächst innerhalb der runden Klammern einen Ausdruck auf seinen Wahrheitswert. Inzwischen sind uns boolesche Ausdrücke bekannt und wir sollten keine Probleme haben, uns hier alternative Bedingungen vorstellen zu können. Wichtig ist lediglich, dass jeder Ausdruck einen Wahrheitswert zurückgibt, da die IF-Bedingung zwingend einen solchen erwartet.

Danach kommt es zur Entscheidung[22]: Liefert der Ausdruck TRUE, so wird der THEN-Block abgearbeitet, der hier einfach TRUE zurückgibt. Andernfalls (z.B. bei `1 > 2`) wird der ELSE-Block ausgeführt und FALSE zurückgegeben. Normalerweise wird in diesen Blöcken Code ausgeführt, der irgendetwas bewirkt, Werte setzt oder sonst etwas Sinnvolles tut. Es ist aber auch wie in diesem Fall möglich, die IF-Anweisung als eine Art Funktion zu benutzen, die einen Rückgabewert produziert. Darauf werden wir erst viel später im Kapitel Funktionen (s. Kapitel I. 6) näher eingehen. Dennoch wäre es möglich, mit einer Variable den Wert „abzufangen".[23]

Wir möchten an dieser Stelle noch mal auf die besondere Gültigkeit von Variablen innerhalb einer Kontrollstruktur hinweisen. Wir hatten dies bereits im Kapitel Namespaces  . Eine lokale Variable, die innerhalb der IF-Anweisung zum ersten Mal initialisiert wird, ist nur innerhalb der IF-Anweisung bekannt. Den umgekehrten Fall, dass wir eine außerhalb definierte Variable nicht im Innern überschreiben wollen, können wir mit dem Befehl **private** lösen. Dies erklären wir aber auch lieber bei Funktionen.

Wir können damit auf einen Schlag eine ganze Menge nützlicher Dinge bewerkstelligen, die vorher nicht möglich waren. Da wir in der Bedingung einen beliebig komplexen Ausdruck auswerten dürfen, sind unserer Kreativität keine Grenzen gesetzt. Folgendes wäre zum Beispiel möglich (daher haben wir so viel Wert auf die logischen Operatoren gelegt):

```
if (count units group player > 4 || alive f1) then {…}
```

Uff, eine Menge neuer Befehle, die wir hier der Vollständigkeit halber und weil sie oft vorkommen, kurz erklären wollen. Hauptaugenmerk liegt aber auf der Verwendung des IF-Befehls!

Zunächst arbeiten wir uns bei solch einem Befehl von innen nach außen. PLAYER ist uns bereits bekannt und gibt die aktuelle Einheit des Spielers zurück. **group** tut genau das, was man erwartet: Es gibt die Gruppe einer Einheit zurück. Wir haben ja gelernt, dass GROUP ein eigener Datentyp in ArmA ist. **units** ist ein sehr nützlicher Befehl, der als Argument eine Gruppe oder eine Einheit erwartet und ein ARRAY zurückgibt, das alle Einheiten der Gruppe enthält. Damit haben wir alles, um den ersten Teil nachvollziehen zu können: Wir fragen ab, ob die Anzahl der Einheiten in der Gruppe des Spielers größer ist als 4. Danach folgt der oder-Operator („||") und ein Befehl, der abfragt, ob die Einheit `f1` noch am Leben ist. Dieser Befehl heißt **alive**. ALIVE erwartet ein OBJECT und gibt einen BOOLEAN zurück. Damit lautet die Bedingung: Sofern die Gruppe des Spielers mehr als 4 Einheiten enthält oder die Einheit `f1` am Leben ist, wird der THEN-Block ausgeführt.

---

[22] Wird ein falscher Datentyp in der Bedingung angeben, kommt es zu einem Fehler oder die Verzweigung wird stillschweigend übersprungen, wenn eine nicht definierte Variable benutzt wird (z.B. bei nil)

[23] Genau dies passiert bei `_retVal = if (1 > 0) then {"It's true"} else {"It's false"};`

An dieser Stelle sollte man sich sehr genau überlegen, wie sich die verschiedenen logischen Operatoren auswirken. In unserem Beispiel gilt die Bedingung als erfüllt, wenn einer der beiden Ausdrücke TRUE ist. Demnach könnte der Spieler sogar tot sein, dennoch würde bei erfüllter zweiter Bedingung die Verzweigung ausgeführt. Für den Fall, dass beide Bedingungen gleichzeitig erfüllt sein müssen, steht uns der und-Operator „&&“ zur Verfügung. Soll stattdessen gewährleistet sein, dass beide Bedingungen nicht zutreffen, muss man die Aussage verneinen, und zwar **jede für sich**. Am einfachsten geht dies mit einem äußeren „!“.

```
if (!(count units group player > 4) && !(alive f1)) then {…}
```
[24]

Damit sollten wir die grundlegende Struktur der IF-Anweisung verstanden haben. Viel mehr gibt es nicht mehr zu sagen. Auf einige Besonderheiten wollen wir euch dennoch hinweisen.

Zunächst erwartet der ein oder andere vielleicht, dass die Verzweigung auch in der bekannten Form `if … elseif … else` zur Verfügung steht. Das ist leider nicht der Fall. Das bedeutet konkret, dass wir nicht mehrere Bedingungen getrennt voneinander angeben können und einen ELSE-Block am Ende haben. In SQF müssen wir für jede Bedingung eine eigene IF-Verzweigung anlegen. Möchten wir also zum Beispiel einen Ausdruck auf 3 verschiedene Werte überprüfen, müssen wir uns quasi mit 2 IF-Anweisungen (oder 3) behelfen:

Listing_18a.sqf
```
1  if (a == 1) then {…}; // Erste Bedingung
2  if (a == 2) then {…}  // Zweite Bedingung
3  else {…};             // Alle anderen Fälle
```

Wir prüfen zunächst, ob `a` den Wert `1` hat. Ist dies nicht der Fall, prüfen wir, ob `a` den Wert `2` hat. Ist auch dies nicht zutreffend, wird der ELSE-Block ausgeführt. Alternativ – wie bereits erwähnt – wäre auch eine dritte IF-Anweisung natürlich möglich gewesen. Wir lernen gleich noch eine alternative Befehlsstruktur kennen, die manchmal etwas übersichtlicher ist.

Dies geht aber nur, weil a nur genau einen Wert annehmen kann. Prüfen wir hingegen mit einer Ungleichung, so haben wir ein Problem mit dieser Variante, wenn `(a > 5)` und `(a < 10)` beides wahr ist (für z.B. `a = 7`) und daher beide IF-Blöcke ausgeführt würden. Hier müssen wir für eine korrekte Nachbildung von `if … elseif … else` Verschachtelung benutzen:

Listing_18b.sqf
```
1  if (a > 5) then {…} // Erste Bedingung
2  else {if (a < 10) then {…} else {…} }; // Zweite verschachtelte Bedingung
```

Mehr dazu im nächsten Kapitel.

Mehrere Verzweigungen – Verschachtelte Bedingungen

Zunächst wollen wir aber noch eine Bemerkung zu verschachtelten Kontrollstrukturen machen: Im Prinzip kann jede Kontrollstruktur, die wir hier beschrieben, beliebig verschachtelt werden. Da Kontrollstrukturen immer in ihren Ausführungsblöcken Code erwarten, darf dieser beliebig komplex sein:

[24] Auch hier sind die Klammern zur besseren Lesbarkeit eingefügt, sie wären tatsächlich aber nicht notwendig.

Listing_19.sqf

```
1  if (side player == west) then // erste Abfrage
2  {
3    if (damage player > 0) then // zweite innere Abfrage
4    {
5      Anweisung1     // im Falle Spieler auf Westseite und verletzt
6    }
7    else { Anweisung2 }    // im Falle Spieler auf Westseite und gesund
8  }
9  else { Anweisung3 };  // Im Falle Spieler auf Nicht-Westseite
```

In dieser schönen Verschachtelung prüfen wir nacheinander die folgenden Bedingungen ab: Zunächst interessieren wir uns für die Seite des Spielers. Ist diese WEST, so gehen wir in den THEN-Block, ansonsten springen wir direkt in Zeile 9. Im THEN-Block fragen wir dann die nächste Bedingung ab, nämlich ob der Spieler verletzt ist. Dies geschieht mit dem Befehl **damage**. DAMAGE erwartet ein OBJECT und gibt NUMBER zurück. Im Falle eines Schadens gehen wir in Zeile 5 und führen `Anweisung1` aus. Ansonsten springen wir zu Zeile 7 und arbeiten `Anweisung2` ab. Wie wir sehen, ist es durchaus wichtig, die ELSE-Blöcke korrekt zu verschachteln! SQF und viele moderne Programmiersprachen beziehen den ELSE-Block immer auf den zugehörigen IF-Block der gleichen Ebene! Daher muss man aufpassen, dass ELSE zum richtigen IF gehört. Folgendes Beispiel würde im Falle eines gesunden Westspielers also gar keine Aktion/Nachricht liefern:

Listing_20.sqf

```
1  if (side player == west) then
2  {
3    if (damage player > 0) then
4    {
5      hint "Spieler verletzt"
6    }
7  }
8  else { hint "Spieler nicht auf West-Seite"};
```

Sofern der Spieler eine West-Einheit ist, aber gesund, kann die innere IF-Anweisung nicht ausgeführt werden. Daraufhin sucht SQF nach einer ELSE-Anweisung und wird in Zeile 8 fündig. Allerdings kommt der Compiler in Wahrheit gar nicht soweit, denn die schließende geschweifte Klammer in Zeile 7 sagt ihm, dass der äußere IF-Block beendet ist. Im Inneren gibt es daher kein ELSE und der Compiler sucht nicht weiter. Außerdem sollte man immer aufpassen, ob das Gegenteil von dem, was man abfragt, wirklich im ELSE-Block steht. Wenn wir die Seite des Spielers mittels „`== west`“ abfragten und im ELSE-Block als HINT „Spieler ist Ost“ ausgeben würden, wäre dies nicht korrekt, denn das Gegenteil von WEST ist nicht WEST und damit alle übrigen Fraktionen, also auch CIVILIAN, RESISTANCE, usw.

Die Verzweigung als switch-case-Bedingung

Nun sind uns also auch Verschachtelungen bekannt. Als letztes wollten wir noch eine alternative Struktur für 100 IF-Zeilen anbieten: die **switch do**-Struktur.

Listing_21.sqf

```
1  switch (_condition) do
2  {
3    case 1:
4    { hint "1" };
5    case 2:
6    { hint "2" };
7    default
8    { hint "default" }
9  };
```

Im Grunde ist SWITCH nichts anderes als die Abkürzung für eine ganze Reihe von IF-Anweisungen. Aber manchmal sind eben Kürze und Übersichtlichkeit es wert, einen neuen Befehl einzuführen.

Auch bei SWITCH prüfen wir zunächst eine Bedingung innerhalb der runden Klammern. Als Signalwort kommt jetzt aber DO und nicht mehr THEN zum Einsatz (was daran liegt, dass wir unabhängig vom Ausdruck immer den gesamten Block zunächst betrachten, geprüft wird erst in den CASE-Zeilen). Anschließend prüfen wir in jeder Zeile einen möglichen Wert von `_condition`, das Schlüsselwort lautet CASE. Statt eines THENs kommt ein Doppelpunkt „:“ zum Einsatz. Der Block danach ist der Code, der ausgeführt werden soll, wenn `_condition` mit dem Wert von CASE übereinstimmt. Der ELSE-Block heißt hier DEFAULT und wird ausgeführt, wenn keine CASE-Zeile zutrifft. Zudem ist er ebenfalls optional, kann also auch weggelassen werden. Damit haben wir die Möglichkeit, eine Variable extrem effizient und übersichtlich auf eine ganze Reihe von möglichen Werten hin zu prüfen.

Achtet bitte immer auf die korrekten Klammern. Einige Strukturen erfordern **geschweifte** Klammern, die generell ausführbaren Code enthalten (WHILE-Schleife), andere erfordern zur Prüfung des Argumentes **runde** Klammern (IF, SWITCH) und **eckige** Klammern haben wir bei Arrays kennengelernt!

Als letzter Hinweis sei angemerkt, dass SWITCH *case-sensitive* für STRINGs ist. Das soll heißen, dass ihr eine Variable natürlich auch auf ihren „Textinhalt“ überprüfen könnt, indem ihr als Prüfargument eine Variable vom Datentyp STRING übergebt und mit CASE dann den Text abfragt. Hier spielt dann aber die Groß- und Kleinschreibung eine Rolle! („Hallo“ ist nicht gleich „hallo“, was man mit *case-sensitive* bezeichnet, also fallabhängig).[25]

B SCHLEIFEN

Mit der Verzweigung sind wir in der Lage, Code zu überspringen oder an eine Bedingung zu knüpfen. Mit anderen Worten: Wir können verschiedene Wege innerhalb des Scriptes gehen. Was ist aber, wenn wir eine Bedingung aktuell überprüfen und sie FALSE liefert, wir aber wollen, dass das Skript erst weiter läuft, wenn sie TRUE wird? Was ist, wenn wir einen bestimmten Code sehr viele Male ausführen wollen? Was ist, wenn wir alle Elemente eines sehr langen ARRAYs abarbeiten wollen? Bei all diesen Problemen hilft uns eine Verzweigung nicht weiter, wir brauchen Schleifen.

Schleifen ermöglichen das mehrfache Ausführen bzw. Abarbeiten eines Code-Fragmentes für eine bestimmte oder unbestimmte Anzahl von Wiederholungen.

Die einfache Zählschleife for

Wir wollen mit der **for**-Schleife beginnen, da sie die einzige Schleife ist, bei der wir konkret angeben, von wo sie starten und bis wohin sie laufen soll. Daher eignet sie sich am besten für die Vorstellung des Schleifen-Konzepts. Es gibt zwei Syntax-Varianten, wir beginnen wie immer mit der etwas sperrigeren, dafür ausführlicheren und letztendlich wartungsfreundlicheren:

[25] Um diese Probleme mit Strings zu lösen, können die Befehle **toLower** und **toUpper** Gold wert sein.

Listing_22.sqf

```
for [{BEGIN}, {CONDITION}, {STEP}] do
{
    STATEMENT;
    ...
};
```

Zur besseren Lesbarkeit wurden die zugehörigen Klammern farbig hervorgehoben. Wie bei all diesen Befehlen ist der Aufbau im Grunde zweigeteilt: Im ersten Teil ([…]) folgt zunächst die Bedingung oder Initialisierung, die die Arbeitsweise der Schleife festlegt. Im zweiten Teil folgt dann die Ausführung des Codes im Falle einer positiven Rückmeldung des Bedingungs-Abschnittes.

Innerhalb des Initialisierungsblockes haben wir drei Anweisungsblöcke ({…}). Im ersten Block (BEGIN) legen wir für eine Laufvariable den Startwert fest. Mit diesem Wert wird die FOR-Schleife starten. Als nächstes folgt die Bedingung (CONDITION), die eine Abbruchbedingung darstellt. Solange die Bedingung TRUE liefert, läuft die Schleife weiter. Sobald aber einmal FALSE zurückgegeben wird, beendet die Schleife ihren Dienst. Als letztes folgt der Schrittweiten-Block (STEP), der festlegt, wie unsere Laufvariable (BEGIN) zu ihrem nächsten höherem Wert kommt. Dies kann durch einfache Erhöhung mit +1[26] passieren, aber auch ein komplizierter mathematischer Ausdruck sein.

Wichtig ist, dass Code-Blöcke, die man immer an den geschweiften Klammern erkennt, entweder Anweisungen, Variablen-Initialisierungen oder ganze Code-Fragmente enthalten können. Damit wollen wir sagen: Innerhalb der geschweiften Klammern kann entweder ein Ausdruck ausgewertet oder eine ganze Code-Zeile bearbeitet werden. Der Schleife ist das egal, solange die Syntax korrekt ist und das heißt hier: die Rückgabewerte vom Typ BOOLEAN sind.

Damit können wir uns eine einfache FOR-Schleife anschauen, die die Zahlen 1 bis 9 ausgibt:

Listing_23.sqf

```
// a loop repeating 10 times
for [{_i=0}, {_i<10}, {_i=_i+1}] do
{
  hint str(_i)
}
```

Wir schauen uns kurz den Code Block für Block an: Zunächst wird innerhalb der FOR-Schleife eine Laufvariable `_i` mit dem Startwert `0` initialisiert. In der Programmierung haben sich `i` und `n` für solche Laufindizes etabliert, man wird ihnen daher sehr häufig begegnen. Wichtig ist hier, dass die Variable `_i` wegen des Unterstriches lokal ist. Damit gilt aber zusätzlich, dass `_i` nur innerhalb der Schleife bekannt ist! Wir haben auf diesen Umstand bereits früher mehrfach hingewiesen. auf `_i` kann nicht außerhalb der Schleife zugegriffen werden! Im nächsten Block legen wir als Abbruchbedingung die Zahl 10 fest. Das ist ganz wichtig! Hier passieren häufig Fehler, weil das Ungleichheitszeichen nicht beachtet wird. Wir hatten gesagt, dass die Schleife abbricht, wenn die Auswertung des Bedingungs-Blockes FALSE liefert. Wenn `_i` also `9` ist und danach um `1` erhöht wird, so ist `_i` beim nächsten Durchlauf gleich `10`. Damit liefert `10 < 10` ein FALSE und die Schleife bricht ab. Wer möchte, dass die Schleife bis `10` geht, muss demzufolge entweder aus `<` ein `<=` machen oder die `10` in eine `11` abändern.

Fehlt nur noch die Schrittweite. Diese wird im letzten Block mit `_i=_i+1` auf `1` festgelegt. Das bedeutet, dass die Schleife am Ende jeden Durchgangs die Laufvariable `_i` genau um eins erhöht.

[26] Siehe Listing 23, es muss sich in jedem Fall um eine Gleichung handeln.

Wir erwähnen noch einmal, dass anstelle der Ausdrücke auch komplexere Code-Fragmente stehen können. Damit ist es also auch möglich, statt der Abbruchbedingung `_i < 10` Folgendes zu schreiben:

```
…, {_i <= count units group player },…
```

Damit würde unsere Schleife von einem nicht erwähnten Startwert solange laufen, bis die Anzahl aller Einheiten der Gruppe des Spielers erreicht ist. Gleiches gilt für den Beginn- und den Schrittweiten-Block.

Es ist vielleicht noch nützlich, darauf hinzuweisen, dass der Schrittweiten-Block wirklich als letztes ausgeführt wird. Wer also die Variable `_i` innerhalb der Ausführung benutzen möchte (z.B. bei einer Textausgabe zur Nummerierung), braucht keine Angst zu haben, dass `_i` vorher um eins erhöht wird. Außerdem ist es möglich, auch negative Schrittweiten anzugeben, um so Schleifen zu erzeugen, die rückwärts zählen:

```
for [{_i=20}, {_i>0}, {_i=_i-2}] do {...}
```
[27]

Zum Schluss wollen wir nicht unerwähnt lassen, dass auch eine Kurzschreibweise der FOR-Schleife existiert:

Listing_24.sqf
```
1  for "VARNAME" from STARTVALUE to ENDVALUE [step VALUE] do
2  {
3    STATEMENT;
4    ...
5  };
```

Diese Variante kommt gänzlich ohne Klammern aus und ist sehr ähnlich zur Sprache VBA. Die Syntax sollte intuitiv verständlich sein. Die Laufvariable wird in Anführungszeichen gesetzt und anschließend wird angegeben, von welchem Startwert bis zu welchem Endwert sie voranschreitet. Die Schrittweite ist standardmäßig auf eins gestellt.[28] Wir raten aber – wie schon oft erwähnt – zur erstbeschriebenen Variante.

Die Prüfschleife while

Damit haben wir eine erste Schleife kennengelernt, die wir allerdings vollständig beschreiben müssen. Damit meinen wir, dass wir sowohl Start- als auch Endwert angeben müssen, damit die FOR-Schleife korrekt arbeitet. Wir können nicht einfach z.B. den Endwert weglassen, nur weil wir ihn nicht wissen. Dafür steht uns ein anderer Schleifen-Typ zur Verfügung: Die **while**-Schleife:

Listing_25.sqf
```
1  while {CONDITION} do
2  {
3      STATEMENT;
4      ...
5  };
```

Zunächst fällt auf, wie einfach die Syntax dieser Schleife ist: Wir haben lediglich – neben den obligatorischen Befehlen WHILE und DO – ein einziges Argument, die Bedingung (CONDITION). Der große Unterschied zur FOR-Schleife ist, dass wir hier eine Bedingung angeben müssen, die vom Rückgabewert ein BOOLEAN ist. Wir müssen also innerhalb der geschweiften Klammern einen Code oder einen Ausdruck angeben, der TRUE zurückliefert, wenn wir möchten, dass `Statement` ausgeführt wird. Ganz wichtig ist dabei ein weiterer, fundamentaler Unterschied der beiden Schleifenarten: Der Code in einer WHILE-Schleife muss in geschweifte Klammern gesetzt werden! Geschweifte Klammern stehen immer um SQF-Code und damit darf neben einer Variablen auch ein komplexer Ausdruck verwendet werden. Somit dürfen wir also auch komplexe Beziehungen abfragen. Dies wollen wir an zwei Beispielen verdeutlichen:

---

[27] Diese Schleife zählt also von 20 beginnend rückwärts, wobei nur gerade Zahlen berücksichtigt werden (20,18,16,…)
[28] Die Schrittweite kann durch den Zusatz STEP im Kopf angegeben werden (`for „_i“ from 0 to 10 step 2 do`).

Listing_26.sqf

```
1  while {alive player} do
2  {
3    if (damage player > 0) then
4    {
5      var_verletzt = true
6    }
7    else { var_verletzt = false }
8  }
```

Mit diesem Skript prüfen wir zunächst in einer äußeren WHILE-Schleife, ob der Spieler am Leben ist. Nur wenn diese Bedingung zutrifft, also TRUE zurückgibt, wird der innere Bereich ausgeführt. Damit läuft das Skript nur solange der Spieler lebendig ist. Bei seinem Tod wird auch die WHILE-Schleife beendet. Innerhalb der äußeren Schleife benutzen wir eine Verzweigung (Zeile 3), um zusätzlich zu prüfen, ob der Spieler einen Schaden erlitten hat. `Damage player` ist standardmäßig 0, gibt aber bei jedem erlittenen Schaden einen Wert zwischen (0;1] zurück. Die Verzweigung führt dazu, dass die globale Variable `var_verletzt` entweder auf TRUE oder FALSE gesetzt wird. Da diese Verzweigung innerhalb einer WHILE-Schleife steht, wird sie nicht nur einmal – wie sonst, wenn sie alleine im Skript stünde – ausgeführt, sondern permanent und fortwährend, solange die WHILE-Schleife aktiv ist!

Listing_27.sqf

```
1  while { driver auto == player || gunner auto == player } do
2  {
3    hint str(speed auto);
4    sleep 2
5  }
```

In diesem Skript prüfen wir zunächst zwei getrennte Bedingungen, die unabhängig erfüllt sein können (||). Der Spieler soll entweder der Fahrer oder der Schütze des Fahrzeuges sein, ausgedrückt durch die Befehle **driver** und **gunner**. Solange der Spieler also Fahrer oder Schütze ist, wird die Geschwindigkeit (**speed**) des aktuellen Fahrzeugs (hier `auto` genannt) ausgegeben.

In Zeile 4 des letzten Skriptes haben wir einen SLEEP-Befehl eingeführt. Dieser ist zunächst merkwürdig, da danach kein weiterer Befehl kommt, wir also quasi zunächst ohne Effekt das Skript für 2 Sekunden „anhalten“. Die Bedeutung liegt in der äußeren WHILE-Schleife. Bisher haben wir über deren Frequenz oder Wiederholungsrate nichts gesagt. Ohne hier jetzt wissenschaftliche Methoden bemühen zu müssen, können wir darauf verweisen, dass eine WHILE-Schleife nicht unendlich schnell aber doch ohne jede weitere Einschränkung deutlich öfter als einmal pro Sekunde ausgeführt wird. In Verbindung mit einem HINT-Befehl würden wir also permanent mehrfach pro Sekunde Nachrichten auf dem Bildschirm ausgeben. Eine WHILE-Schleife wird wahrscheinlich nicht unendlich oft durchlaufen – es gibt Meinungen, nach denen nach ca. 10.000 Durchläufen Schluss ist. Für uns ist dieses Wissen aber eher akademischer Natur. Wichtig ist allein, ob die WHILE-Schleife mehrmals pro Sekunde, also in ihrer Standardkonfiguration, eine Bedingung prüfen soll, oder ob es auch aller 10 Sekunden ausreichend ist. Je nach dem stellen wir z.B. die Frequenz per SLEEP-Befehl ein. Ganz besonders wichtig sind diese Überlegungen bei performancehungrigen Aktionen oder bandbreitenbelastenden Befehlen wie PUBLICVARIABLE[29].

An dieser Stelle sei auch der Hinweis gegeben, dass die WHILE-Schleife komplett übersprungen wird, wenn die Bedingung bei der erstmaligen Prüfung bereits FALSE ist. Im Gegensatz zu DO-WHILE-.Schleifen haben wir in ArmA oder SQF keine Möglichkeit, einen Schleifendurchgang auf jeden Fall einmal zu erzwingen und die Überprüfung an den Schluss zu setzen.[30]

[29] Wir verweisen hierfür auf den zweiten Teil dieses Handbuches: Teil II – SQF im Multiplayer.
[30] Wir könnten diesen Umstand z.B. durch den Befehl WAITUNTIL erzwingen, das würde aber das ganze Skript anhalten.

Die Warteschleife waitUntil

Man kann sich streiten, ob **waitUntil** eine Schleife ist – sie ist es aber und allemal wichtig zu kennen, daher stellen wir sie jetzt vor. Das schöne ist: Sie ist extrem einfach zu benutzen und sehr mächtig.

Description

Description:	Suspend execution of function or SQF based script until condition is satisfied.

Syntax

Syntax:	**waitUntil** condition
Parameters:	condition: Code
Return Value:	Nothing

Die Syntax dieses Befehls ist, wie versprochen, sehr einfach. Es bedarf lediglich des Befehls WAITUNTIL sowie der zu prüfenden Bedingung (CONDITION). Aufpassen müssen wir beim Parameter: Dieser ist vom Type Code. Dies hat eine wichtige Konsequenz für die Klammerart, die wir verwenden müssen! Da Code generell ausführbar ist und nicht nur aus Befehlen sondern auch auszuwertenden Code-Fragmenten bestehen kann, muss die Bedingung folglich wie ein Code-Block behandelt werden und Blöcke stehen in SQF – wie wir bereits wissen! – immer in geschweiften Klammern. Damit können wir die Syntax wie folgt angeben:

```
waitUntil {Condition; ...};
```

Bewusst auch hier wieder beide Semikola ausgeschrieben[31]: Im Inneren können mehrere eigenständige Code-Fragmente stehen, die mit ;: beendet werden sollen, außerdem ist WAITUNTIL eine Anweisung, die wir ebenfalls gewohnheitsgemäß mit einem ; beenden, da sie logischerweise nicht die letzte Anweisung ist.

WAITUNTIL ist eine Schleife, die das Skript solange verzögert, bis die angegebene Bedingung TRUE zurückliefert. Damit ist es ein sehr mächtiges Konstrukt, das auf einfache Weise komplizierte Abläufe ermöglicht. Denn WAITUNTIL wartet wirklich solange, bis die Bedingung eintritt und solange passiert auch nichts anderes. Eine einfache IF-Verzweigung, die wir bereits kennengelernt haben, würde entweder ausgeführt oder nicht ausgeführt, aber sie würde das Skript als Ganzes nicht anhalten! WAITUNTIL arbeitet hier anders. Um dem geneigten Leser die Arbeitsweise zu verdeutlichen, hier eine Möglichkeit, WAITUNTIL nachzuprogrammieren:

```
while {!(condition)} do {};
```

Im Grunde handelt es sich also nur um eine WHILE-Schleife, die überprüft, ob die Bedingung, die eigentlich TRUE sein soll, aktuell noch FALSE ist. Da der ausführende Teil leer ist, macht diese Schleife nichts. Dennoch wird das Skript solange verzögert, bis CONDITION TRUE ist, denn eine WHILE-Schleife wird nicht übersprungen, sondern immer wieder erneut durchlaufen.

Für einen ausführlichen Einblick in die Verwendungsmöglichkeiten von WAITUNTIL verweisen wir auf das Kapitel „Vorzeitiges Beenden von Schleifen und Skripten - Sprungstellen“.

Die Durchlaufschleife forEach

Jetzt kommen wir zu einem ganz besonders mächtigen Schleifentyp – und dem wohl vielseitigsten in ArmA bzw. SQF. Mit **forEach** lässt sich einerseits fast jedes Problem lösen und andererseits sehr elegant und kurz programmieren. Daher schauen wir uns zunächst die Syntax an:

[31] Bitte für dieses etwas heiklere Thema in Kapitel I. 4.B nachlesen.

Description

Description:	Executes the given command(s) on every item of an array. The array items are represented by _x. The array indices are represented by _forEachIndex. In ArmA2 & VBS2, the variable _x is always local to the forEach block so it is safe to nest them.

Syntax

Syntax:	command **forEach** array
Parameters:	command: String array: Array
Return Value:	Nothing

Diesmal müssen wir der Erklärung (Description) etwas mehr Beachtung und Sorgfalt widmen. Zunächst lesen wir, dass die Schleife einen gegebenen Befehl (oder mehrere) für jedes Element eines ARRAYs ausführt. ARRAYs sind uns bereits aus Kapitel I. 4.C bekannt. Zur Wiederholung: Es handelt sich bei ihnen in SQF um Datenfelder, die mehrere Elemente speichern können und den Zugriff über einen wohldefinierten Index erlauben. ARRAYs werden durch eckige Klammern initialisiert und können verschiedenste Datentypen aufnehmen sowie mehrdimensional aufgebaut sein.

Der zweite Satz verrät uns, wie wir innerhalb der FOREACH-Schleife auf das aktuelle Element des ARRAYs zugreifen können: mittels **_x**. Dabei handelt es sich um eine sogenannte MAGIC VARIABLE, die wir aber erst später in Kapitel I. 9. behandeln wollen. Dies kleine unscheinbare _X spielt eine sehr wichtige Rolle, wie wir gleich sehen werden. Schließlich wird uns noch erklärt, dass _X immer lokal für den FOREACH-Block ist, also nicht außerhalb der Schleife genutzt werden kann, was wir aber soweit bereits kennen (s. Namespaces).

Für falsch halten wir aber die Parameterbeschreibung des Arguments COMMAND. Dieser ist kein String, sondern wie bei den anderen Schleifen vom Type CODE und damit muss er in geschweifte Klammern eingeschlossen werden. Hier ist also manchmal etwas Vorsicht geboten. Die angegebenen Beispiele sind aber korrekt und spiegeln diese Tatsache auch wider.

Damit können wir uns ein einfaches Beispiel anschauen:

Listing_28.sqf

```
1  _array = [unit1, unit2, unit3];
2  {
3    _x setDamage 1
4  } forEach _array
```

Dieses sehr hilfreiche Beispiel verdeutlicht, wie viel Schreibarbeit nötig wäre, wenn die FOREACH-Schleife nicht zur Verfügung stünde. Zunächst definieren wir ein lokales ARRAY `_array`, das z.B. aus wichtigen eigenen oder feindlichen Einheiten besteht. Anschließend möchten wir, dass diese Einheiten zerstört werden. Wir kennen bereits den Befehl DAMAGE, um den Schaden einer Einheit abzufragen. Getreu guter Programmierkonventionen heißen Methoden, die eine Eigenschaft eines Objektes verändern, immer SETMETHODENNAME[32], also in unserem Fall **setDam(m)age**.

[32] Dieser Hinweis ist für den Anfänger ganz besonders nützlich. Sucht ihr einen Befehl, der eine Eigenschaft verändern soll, solltet ihr also z.B. mit dem ArmA2 Command Lookup zunächst immer SET eingeben. Der gesuchte Befehl könnte dabei sein.

Wir eröffnen also einen Code-Block mit geschweiften Klammern (unser COMMAND-Argument), setzen dahinter den Befehl FOREACH und zum Schluss das zweite, notwendige ARRAY-Argument. Damit haben wir das Grundgerüst bereits fertig: Der Code innerhalb des Blockes wird für jedes Element des ARRAYS `_array` ausgeführt. Der Code selbst ist denkbar einfach: Jedem aktuell angewählten Element (`_x`!) wird mit dem Befehl SETDAMAGE der Schaden 1 zugewiesen (1 ist 100%).

Für etwas versiertere (oder interessiertere) Leser hier die Möglichkeit, die FOREACH-Schleife mit den bisher bekannten Mitteln nachzubauen:

Listing_29.sqf

```
1  for [{_i=0}, {_i < count _array}, {_i=_i+1}] do
2  {
3     (_array select _i) setDamage 1
4  }
```

Wir sehen also: Die FOREACH-Schleife ist nichts anderes als eine FOR-Schleife, die alle Elemente einzeln durchgeht. Da der Code immer gleich ist, weil wir die obere Grenze stets ganz dynamisch ohne Kenntnis der eigentlichen Länge mit `count _array` bestimmen können, kann man dies sozusagen fest als neue Schleife mit neuem Namen speichern und bekommt – et voilà – die FOREACH-Schleife.

Die FOREACH-Schleife wird genau einmal durchlaufen und iteriert dabei über alle Elemente des angegebenen ARRAYS. Selbstverständlich kann man jetzt aus den Vollen des bisher Gelernten schöpfen und die Schleifenarten beliebig miteinander kombinieren. Brauchen wir zum Beispiel während einer Mission permanent eine Schleife über ein ARRAY, um z.B. die Lage von Einheiten durchgängig abzufragen und zu markieren, so hilft uns die FOREACH-Schleife allein nicht weiter, wir müssen eine äußere WHILE-Schleife nutzen und im Inneren unsere FOREACH-Schleife platzieren, die wir dann natürlich mit einer IF-ELSE-Bedingung noch weiter verfeinern können. Wir sehen – grenzenlose Möglichkeiten.

An dieser Stelle sei auf eine Besonderheit hingewiesen, die sich eigentlich nur sehr schwer erschließt: Die FOREACH-Schleife kann auch ohne `_x` genutzt werden, also mit einem Code, der nicht vom aktuellen Element abhängt. Wozu das gut sein soll? Seht ihr hier:

Listing_30.sqf

```
1  {
2     player addMagazine "30Rnd_556x45_Stanag"
3  } forEach [1,2,3,4];
```

Tja, was tut dieser Code? Er fügt dem Spieler vier M16-Magazine hinzu. Vier? Ganz genau. Warum nicht 10? Das ist eine gute Frage. Denkt mal darüber nach, Auflösung gibt es in der Übung [1]! Jedenfalls eine sehr praktische Anwendung dieses Schleifentyps, da damit Ausrüstungsskripte deutlich kompakter geschrieben werden können.

Zum Schluss möchten wir euch unbedingt noch auf ein paar vordefinierte ARRAYS von ArmA hinweisen, die euch erst einige sehr nützliche Skriptmöglichkeiten in Verbindung mit FOREACH-Schleifen eröffnen.

Wichtige vordefinierte Arrays

SQF bietet bereits in seinem Sprachumfang einige vordefinierte Befehle, die euch ARRAYS als Rückgabewerte liefern. Da wir aber Rückgabewerte als solche noch nicht hatten (wir verweisen erneut auf das Kapitel I. 6 Funktionen), reicht es völlig, wenn ihr euch diese quasi-Befehle als feste Bezeichner für ARRAYS merkt. Also Befehl gleich ARRAY. Ist nicht ganz korrekt, in der Praxis aber egal. Jeder der folgenden Befehle ist also als Argument für eine FOREACH-Schleife geeignet.

```
{ Anweisung; ... } forEach allUnits/allDead/playableUnits/vehicles/…;
```

allUnits. Dieses Array beinhaltet alle aktuellen Einheiten der Mission, also alle Infanteristen, Fahrzeuge und Flugzeuge[33], alle Zivilisten und alle Einheiten der eigenen Seite, aber keine toten Einheiten. Mit dem erweiterten COUNT-Befehl lässt sich so ganz einfach die aktuelle Feindanzahl bestimmen:

```
{side _x == EAST} count allUnits;
```

Dieses Beispiel sollte man sich ruhig etwas länger ansehen, denn hier passiert etwas sehr Magisches. Der Befehl COUNT zählt an sich einfach die Elemente eines ARRAYs. Nun haben wir vor den Befehl aber noch einen Code-Block geschrieben, den wir so an sich bei einer FOREACH-Schleife erwartet hätten. Es handelt sich um eine spezielle Kurzschreibweise, die man einfach kennen muss. Der Code-Block gibt also an, welche Elemente im ARRAY gezählt werden sollen. Haben wir einen Spieler und zwei Feinde auf der Karte, wäre `count allUnits` gleich 3. Mit der obigen Einschränkung lautet das Ergebnis hingegen nur 2. Alternativ gibt es auch hier einen anderen Befehl: **countEnemy**. Klingt im ersten Moment gut, würde aber wahrscheinlich als Ergebnis 0. Warum? Weil COUNTENEMY geringfügig anders arbeitet. Es zählt nämlich nur die Feinde, die aktuell im angegebenen ARRAY dem Spieler als feindlich bekannt gelten! Und wir können euch sogar noch eine dritte mögliche Alternative anbieten, die diesmal wieder das richtige Ergebnis liefert:

```
east countSide allUnits;
```

Das Beispiel kann zur eigenen Übung gerne eigenständig nachvollzogen werden.[34]

allDead. Enthält eine Liste aller toten Einheiten, sowohl in als auch außerhalb von Fahrzeugen. Kann ganz nützlich sein, um z.B. bestimmtes Equipment von Leichen zu entfernen oder hinzuzufügen oder auch, um Leichen verschwinden zu lassen.

playableUnits. Ein sehr guter Befehl für den Multiplayer, da er ausschließlich Einheiten enthält, die im Editor auf „spielbar“ gesetzt wurden. Somit kann gezielt Ausrüstung zugewiesen oder ein Skript ausgeführt werden. Ebenso eine schnelle Methode, um die Anzahl der verbliebenen Spieler zu ermitteln (AI zählt aber hier auch dazu!)[35].

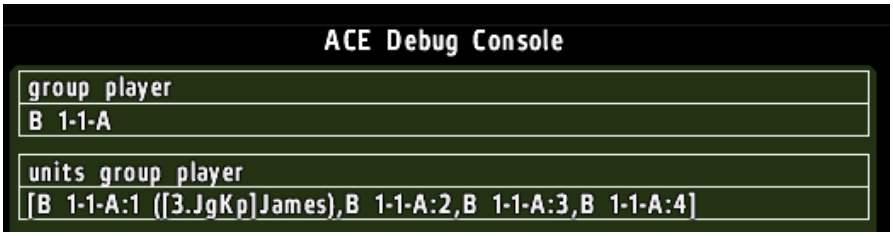
vehicles. Prinzipiell ein ebenso nützlicher Befehl wie ALLUNITS, nur dass er alle Fahrzeuge der Mission enthält (leer als auch besetzt). Leider etwas außer Mode gekommen, da er unter ACE nicht mehr so einfach funktioniert, da auch die Benzinkanister mitgelistet werden. Kann man aber umgehen bzw. geschickt austricksen  . Ach, machen wir daraus Übung [3]!

list. Ein praktischer Befehl, um alle Einheiten eines Triggers zu erhalten, die ihn auslösen würden. Ganz wichtig ist hier „würden“. Also nicht die Einheiten, die ihn ausgelöst haben, sondern schon vorher, die also potenziell auslösen könnten. Daher liefert LIST z.B. alle Einheiten, die einen BLUEFOR-Trigger auslösen, auch wenn dieser nur auf einmalig steht.

units. Gibt alle Einheiten einer Gruppe als Array zurück. Also sehr praktisch, wenn man für alle Mitglieder eine Gruppe ein Skript ausführen möchte. Dabei erwartet UNITS als Argument eine Einheit oder eine Gruppe. Wir erinnern daran, dass GROUP ein eigener Datentyp ist! Folgender Screenshot verdeutlicht beide Befehle. Während GROUP quasi die Bezeichnung der Gruppe liefert (B 1-1-A), liefert UNITS die zugehörigen Einheiten (B 1-1-A: unit):

---

[33] Korrekt müsste man sagen, dass alle Infanteristen innerhalb und außerhalb von Fahrzeugen gezählt werden. Fahrzeuge zählen also nicht extra. Das wollen wir doch glatt in der Übung [2] einmal als Aufgabe stellen!
[34] Ein Blick in ArmA2 Command Lookup verrät: Es gibt COUNTENEMY, COUNTFRIENDLY, COUNTSIDE, COUNTTYPE, COUNTUNKNOWN.
[35] Dieser Befehl ist im SP leider nicht verfügbar, daher kann man ihn auch nicht im Editor testen! Daher besondere Vorsicht, wenn er bei Skripten eingesetzt wird, die lokal getestet werden. Hier mit einem Workaround ALLUNITS arbeiten.



Übungen

[1] Wir wollen zunächst unsere Schuld begleichen und die Antwort auf die Frage liefern, warum folgender Code 4 Magazine hinzufügt, oder seid ihr selbst darauf gekommen?

```
{player addMagazine "30Rnd_556x45_Stanag"} forEach [1,2,3,4];
```

[2] Eine zweite Aufgabe wurde bereits in der Fußnote angedeutet: Ihr habt mehrere Möglichkeiten kennengelernt, feindliche Einheiten zu zählen. Wir möchten jetzt ein Skript schreiben, dass uns die Anzahl der aktuell aufgesessenen Einheiten nennt. Wie sähe der Code dafür aus?

Hinweis: Ihr braucht dafür den erweiterten COUNT- sowie den **vehicle**-Befehl.

[3] Wir hatten die Problematik des Befehls VEHICLES besprochen. Dieser zählt leider neben allen Fahrzeugen auch Benzinkanister mit. Aber ihr könnt dieses Problem lösen, indem ihr den erweiterten COUNT-Befehl benutzt und alle Benzinkanister über **typeOf** ausschließt. Bekommt ihr das hin?

[4] Eine Aufgabe für IF-Verzweigungen: Wir wollen den Abstand der Spieleinheit zu einem Fahrzeug bestimmen. Unterschreitet der Abstand einen bestimmen Grenzwert, so möchten wir, dass das Fahrzeug explodiert, also zerstört wird. Könnt ihr dies – im Ansatz – als Skript umsetzen?

Hinweis: Für den Abstand könnt ihr den einfachen Befehl **distance** verwenden.

[5] Wir erweitern Aufgabe [4] um folgende Ergänzung: Wir möchten jetzt, dass das Auto immer explodiert, wenn sich ihm eine beliebige Person nähert. Könnt ihr auch dafür eine Lösung finden?

Hinweis: Dazu könnt ihr den Ansatz mit einer FOREACH-Schleife wählen.

[6] Wir erweitern Aufgabe [5] noch einmal um folgende Ergänzung: Wir möchten, dass das Auto nur explodiert, wenn es zusätzlich einen gewissen Schaden erlitten hat. Nach der Explosion des Fahrzeugs möchten wir außerdem eine Meldung ausgeben, die uns darüber informiert, wie stark der eventuell erlittene Schaden ist. Damit solltet ihr dann wirklich etwas gefordert sein (für alle, die sich bis hierher gelangweilt haben).

[7] Und um das ganze wirklich Praxistauglich zu machen: Der Code aus Aufgabe [6] ist soweit fertig, aber er wird nur einmal ausgeführt. In einer Mission möchtet ihr aber eventuell ein solches Fahrzeug ja von Anfang an als Falle aufstellen und nicht das Skript nur ein einziges Mal ablaufen lassen. Passt daher das Skript aus der vorherigen Aufgabe so an, dass es von Missionsbeginn an so lange läuft, bis das Fahrzeug explodiert ist.

Hinweis: Die Lösung liefert eine WHILE-Schleife

I. 6. FUNKTIONEN

Ab jetzt betreten wir fortgeschrittenes Terrain! Das heißt aber auch – sofern ihr bis hierher durchgehalten und mitgearbeitet habt – dass ihr euch bereits das grundlegende Wissen und ein solides Fundament erarbeitet habt! Glückwunsch dazu. Wir können euch ohne Bedenken damit auf eigene Projekte und die ArmA-Welt im Allgemeinen loslassen. Dennoch werdet ihr euch an einigen Problemen die Zähne ausbeißen oder bestimmte Probleme gar nicht erst angehen können, weil euch – wie es so schön heißt – das Insider-Wissen fehlt. Ein zentraler Aspekt beim fortgeschrittenen Programmieren mit SQF sind Funktionen und das ist Thema dieses Kapitels.

EINE FUNKTION IST EINE ABGESCHLOSSENE CODE-EINHEIT, DIE IM GEGENSATZ ZU ANWEISUNGEN EINE BERECHNUNG ODER VERARBEITUNG DURCHFÜHRT UND IN DER LAGE IST, DAS ERGEBNIS ZURÜCKZUGEBEN. FUNKTIONEN VERFÜGEN DAHER ÜBER ÜBERGABERARAMETER UND RÜCKGABEWERTE.

Wer bis hierhin aufmerksam mitgelesen hat, der weiß, dass wir verschiedentlich von Rückgabewerten gesprochen haben, ohne darauf näher einzugehen. Oftmals sind die bisher kennengelernten Befehle auch Funktionen mit einem Rückgabewert, das heißt sie sind in der Lage, neben ihrer eigentlichen „Aufgabe“ z.B. Fehler oder Hinweise zurückzugeben, die Rückschlüsse über den Verlauf des Skriptes geben. Dazu zwei Beispiele:

```
_status = isNil "var_a";
```

Diesen Befehl kennen wir bereits: ISNIL prüft, ob eine Variable zum jetzigen Zeitpunkt schon initialisiert wurde, also bekannt ist. Wir können ISNIL direkt z.B. in einer IF-Bedingung benutzen, aber wir können den Rückgabewert des Befehls ISNIL (oder eben der Funktion) auch in einer Variable zwischenspeichern, wenn wir ihn andernorts noch benötigen. `_status` ist also nach der Ausführung entweder mit TRUE oder FALSE belegt, den einzigen beiden Werten des Datentyps BOOLEAN. Im Vergleich zu ausführenden Skriptbefehlen wie HINT, SLEEP oder SETDAMAGE, die keinen Rückgabewert besitzen, geben Funktionen wie ISNIL, COUNT oder alle GET-Befehle wie DAMAGE, ALIVE, GETPOS einen Wert zurück. Der Datentyp dieses Rückgabewertes kann dabei beliebig sein und alle Datentypen abdecken, die wir in Kapitel I. 4.C behandelt haben. Jetzt zum zweiten Beispiel:

Listing_31.sqf

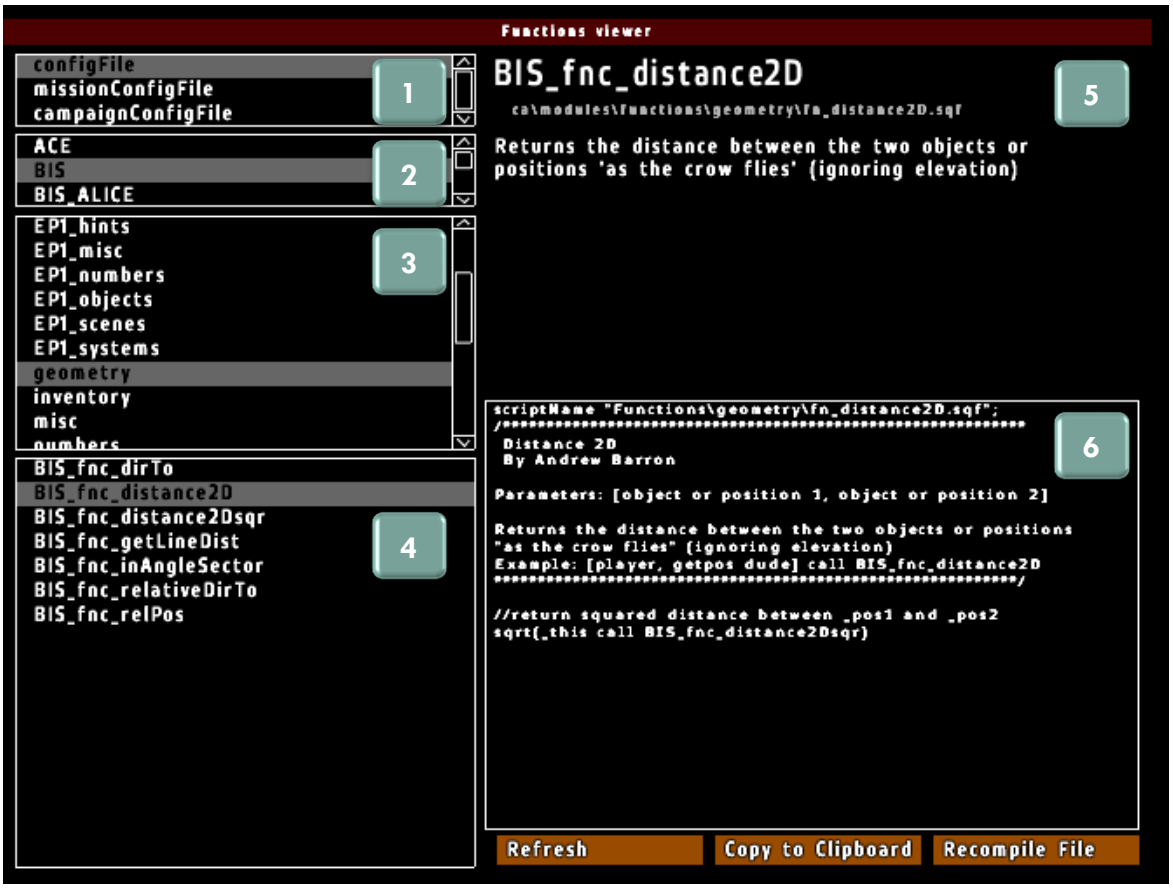
```
1  _a = 6; _b = 5;
2  _status = if (_a < _b) then
3  {
4    hint (str(_a) + " ist kleiner als "+str(_b));
5    "wahr"
6  }
7  else
8  {
9    hint (str(_a) + " ist größer als "+str(_b));
10   "falsch"
11 }
```

Vom Inhalt her ist hier nichts Neues hinzugekommen. Wir initialisieren zwei Variablen vom Typ NUMBER in Zeile 1 und fragen dann mit der IF-Bedingung in Zeile 2 ab, welcher Wert der größere von beiden ist. Abhängig vom Ergebnis wird entweder der THEN-Teil (Zeile 4-5) oder der ELSE-Teil (Zeile 9-10) ausgeführt. Was wirklich neu ist, sind die bloßen STRINGS „wahr“ und „falsch“. Diese hängen scheinbar etwas zusammenhanglos „in der Luft“. Zudem haben wir die IF-Bedingung einer Variable `_status` zugewiesen, was wir bisher ebenfalls noch nie getan haben. Die Auflösung erhalten wir, indem wir uns die Variable `_status` einmal per HINT-Befehl ausgeben lassen. Wir werden dann die Ausgabe „wahr“ oder „falsch“ erhalten, also eben einer jener beiden STRINGS, die am Ende der jeweiligen Verzweigung stehen. Wir haben also eine zusätzliche Option der IF-Bedingung ausgenutzt, die wir bisher nicht berücksichtigt haben: Die IF-Bedingung gibt jeweils den Rückgabewert der Verzweigung zurück. Dazu müssen wir aber noch wissen, was das genau ist, was uns zur nächsten Definition bringt:

Eine Funktion wird eingeleitet durch einen Bezeichner, gefolgt von ihrem Rumpf {...}.
Eine Funktion wird strikt in SQF-Syntax geschrieben.
Der Rückgabewert einer Funktion ist stets der letzte Ausdruck innerhalb der Funktion.

Im Grunde sind Funktionen nichts anderes als Skripte, um nicht zu sehr ins Theoretische abzuschweifen. Warum wir dennoch so viel Wert auf Funktionen legen, liegt an zwei wichtigen Unterschieden: Es gibt viele vorgefertigte Funktionen von ArmA2, ACE, CBE und anderen Modifikationen, die uns das Leben stark erleichtern und wir können uns Funktionen selber schreiben, um bestimmte, sich widerholende Arbeitsschritte eleganter zu lösen und vor allem, um Performance zu sparen.

Wir wollen mit den vorgefertigten Funktionen beginnen, da der Aufwand zur Benutzung minimal ist und auf einen Schlag völlig neue Möglichkeiten zur Verfügung stehen. Zunächst wollen wir euch daher ein Hilfsprogramm vorstellen, dass in ArmA implementiert ist und unschätzbare Hilfe leistet: der **Funktions-Viewer**.



Zum Aufruf dieses Viewers platziert ihr euch einfach eine Einheit im Editor sowie das Funktionsmanager-Modul. Danach könnt ihr folgende zwei Befehle per Init-Zeile, Auslöser oder Funkauslöser aufrufen:

Listing_32.sqf

```
1  waituntil {!isnil "bis_fnc_init"};
2  [] call BIS_fnc_help
```

Lustigerweise rufen wir hier bereits eine Funktion auf, denn mit CALL wird eine Funktion aufgerufen. Dazu aber später mehr. Jetzt interessieren wir uns vorrangig dafür, wie wir den Viewer benutzen. Das ist zum Glück sehr einfach: Wenn wir eine bestimmte Funktion suchen, dann stellen wir im ersten Feld configFile ein, im zweiten Feld das gewünschte Addon oder Modul ein, in dem die Funktion zu finden ist (BIS ist also ArmA), suchen dann im dritten Feld die Kategorie heraus und bekommen dann letztendlich im vierten Feld alle zur Verfügung stehenden Funktionen aufgelistet. Haben wir dort eine Funktion selektiert, liefert uns das fünfte Feld eine kurze Erklärung und ein Verwendungshinweis und Feld sechs präsentiert uns sogar noch den

gesamten Code. Für eigene Zwecke nutzen können wir die Funktion, indem wir sie z.B. mit dem mittleren Button in die Zwischenablage kopieren oder eben per CALL/SPAWN aufrufen.

Dann wollen wir uns jetzt zwei Beispiele dazu ansehen und den interessierten Leser dann selbstständig mit dem Viewer auf Entdeckungsreise gehen lassen.

Wir schauen uns zunächst die Funktion BIS_fnc_Distance2D an. Diese finden wir unter `BIS >> geometry`. Die Hilfe in Feld 6 verrät uns, dass die Funktion folgendermaßen aufgerufen wird:

```
Parameters: [object or position 1, object or position 2]
Example: [player, getpos dude] call BIS_fnc_distance2D
```

Bevor wir den Aufruf selber durchführen, sind wohl ein paar Worte zum Befehl **call** angebracht.

Description

Description:	Executes the function string. The *argument(s)* (if any) are passed as _this. (*argument(s)* are passed in an array). To execute a sleep function in the called code, execute it with spawn instead.

Syntax

Syntax:	argument(s) **call** body
Parameters:	argument(s): Any Value - Optional. Argument that is passed to the function in the "_this" variable. body: Code - A function body provided directly 'inline' or the String returned from the commands loadFile or preprocessFile.
Return Value:	Anything - The last value given in the function is returned. See the topic Function for more information.

CALL ist zunächst ein Befehl, der einen STRING quasi als Funktion ausführt. Wir können der Funktion, die wir mit CALL aufrufen, Argumente voranstellen. Dies kann ein einziges Argument sein oder ein ARRAY mit mehreren Parametern. Innerhalb des Skriptes können wir darauf mit der magischen Variable _THIS zugreifen. Dazu aber später mehr. Zum jetzigen Zeitpunkt reicht es aus, wenn wir verstehen, dass wir eine vordefinierte Funktion mit CALL aufrufen können. **spawn** arbeitet auf die gleiche Weise wie CALL, mit dem Unterschied, dass SPAWN immer einen neuen Thread für die Funktion eröffnet, die Funktion also quasi immer in einer neuen Datei ausführt und nicht am Ort des Aufrufes selber.[36] Wir wollen nun endlich unsere Funktion aufrufen. Dazu folgen wir dem obigen Beispiel. Wir hatten vorher bereits ein Übungsskript, indem wir mit DISTANCE den Abstand zwischen der Spielereinheit und einem Fahrzeug namens `auto` abgefragt haben. Dies wiederholen wir jetzt mit der Funktion:

Listing_33.sqf

```
1  _distanz = [player, auto] call BIS_fnc_distance2D;
2  hint str(_distanz)
```

Damit haben wir alles nötige bereits getan und die Distanz wird uns auf dem Bildschirm ausgegeben. Wer beide Varianten gegenüberstellt, wird feststellen, dass der Befehl DISTANCE nicht dasselbe liefert wir die Funktion BIS_fnc_Distance2D. Dies wird an unterschiedlichen Berechnungsmodellen liegen.

Ein wenig Stöbern liefert z.B. die Funktion BIS_fnc_halo unter `BIS >> EP1_misc`. Diese wird gestartet mit

```
(player) call BIS_fnc_halo;
```

Der Effekt ist sicherlich sehr lustig, wenn auch etwas ungewollt. Gedacht ist dies für den Absprung aus einem Flugzeug, aber immerhin, auch das funktioniert. Wer die korrekte Höhe dazu haben möchte, braucht nur das Argument um die Höhe als ARRAY erweitern:

[36] Allein über diese zwei Befehle kann man Kapitel füllen…wir vertrösten den geneigten Leser auf später.

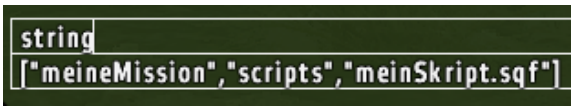
```
[player, 5000] call BIS_fnc_halo;
```

Jetzt versucht einmal ohne diese Funktion den Effekt selber zu Skripten!

Diese beiden Beispiele waren jetzt sehr spielbezogen, der Wert der Funktionen liegt aber gerade in ihrer Unterstützung für komplexe Skriptaufgaben. So gibt es z.B. eine Funktion CBA_fnc_split unter `CBA >> Strings`. Diese Funktion zerlegt eine Zeichenkette je nach angegebenem Separator in ihre Einzelteile. Damit kann z.B. eine Pfadangabe zu einem Skript auf das Skript alleine reduziert werden. Oder ein Wort kann in seine einzelnen Buchstaben zerlegt werden, um dessen Länge zu ermitteln:

```
_string = ["meineMission\scripts\meinSkript.sqf", "\"] call CBA_fnc_split;
```

Das Ergebnis sieht wie folgt aus:



Wir haben also erfolgreich die Pfadangabe getrennt, indem wir als Separator „\“ angegeben haben. Und noch das Beispiel für die Wortlänge:

Listing_34.sqf

```
1   hint str(count ([str(name player)] call CBA_fnc_split))
```

Wir haben auf eine Hilfsvariable verzichtet und das Ergebnis der Funktion direkt per HINT-Befehlausgeben lassen. Diesmal haben wir als String den Namen des Spielers übergeben und anschließend dessen Länge ausgegeben. Wie so oft kommt es hier auf eine saubere *Klammerarbeit*[37] an  .

Damit möchten wir diesen Bereich wieder verlassen. Wir hoffen, dass dieser kurze Einblick verständlich und nachvollziehbar war und ihr den Wert von Funktionen wenigstens im Groben einschätzen könnt. Die Möglichkeiten sind schier endlos und es kommen immer neue Funktionen hinzu. Wichtig ist aber an dieser Stelle nur, dass ihr verstanden habt, was Funktionen sind und wie ihr sie benutzen könnt, wenn ihr denn eine brauchen solltet. Alles andere kommt dann sowieso mit der Zeit und mit der Erfahrung.

Wir wollen zum Abschluss dieses Kapitels wenigstens eine Funktion aber auch selber geschrieben haben, denn darin liegt – wie bereits angedeutet – der zweite große Nutzen von Funktionen. So kann es euch ja durchaus passieren, dass ihr bei größeren Projekten (man wird ja noch träumen dürfen?) einige Skriptaufgaben mehrfach ausführen müsst. Dies könnte im einfachsten Falle ja z.B. eine Maximum-Funktion sein, die euch einfach den größeren von zwei Werten zurückgibt. Dieses Beispiel wollen wir uns jetzt erarbeiten. Dabei werden wir allerdings ein Konzept brauchen, das sich „magic Variable“ nennt und auf das wir gesondert noch einmal im Kapitel Magic Variables eingehen.

Auch ohne Funktion wäret ihr mit dem jetzigen Wissen in der Lage, die geforderte Aufgabe mit SQF zu lösen, es handelt sich ja nur um eine ganz einfache Verzweigung:

```
if (a > b) then {ergebnis = a} else { ergebnis = b }
```

Soweit so gut. Wir prüfen zwei Werte auf Ungleichheit und definieren eine Variable `ergebnis`, die den größeren Wert speichert. Strenggenommen entgeht uns dabei der Fall der Gleichheit, dies wäre aber für den geforderten Fall nicht tragisch, da im Falle der Gleichheit beide Werte ja als Rückgabewert in Frage kommen. Unschön an dieser Variante sind nun zwei Dinge: Zum einen benötigen wir eine Hilfsvariable `ergebnis`, denn bisher hatten wir ja keine Möglichkeit, das Ergebnis einer IF-Bedingung abzufragen und zum anderen müssten wir für jedes Mal, bei dem wir diese Funktion brauchen, die IF-Bedingung neu formulieren. Beides löst sich in Wohlgefallen auf, wenn wir eine eigene Funktion definieren.

---

[37] Achtung bei diesem Skript! Die runden Klammern nach COUNT sind nicht nur für die Ästhetik! Ohne geht es nicht, da hier eine spezielle Reihenfolge vorliegt, die der Compiler nicht von alleine korrekt versteht.

Listing_35.sqf

```
1  fnc_max =
2  {
3    _a = _this select 0;
4    _b = _this select 1;
5    if (_a > _b) then {_a} else {_b}
6  }
```

Das war's. Die Zeilen 3 und 4 sind momentan leider noch nötig, damit das Ganze funktioniert, wir möchten aber die Erklärung auf das nächste Kapitel verschieben. Im Grunde lesen wir mittels _THIS immer die übergebenen Parameter aus. Die IF-Verzweigung ist diesmal wunderschön kompakt, da wir lediglich von der Funktion den größeren Wert als Rückgabewert erwarten. Dies geschieht nach der Definition zu Beginn dieses Kapitels immer durch den letzten Ausdruck. Ein Ausdruck ist dabei kein Befehl sondern ein Rückgabewert, also in unserem Falle `_a` oder `_b`. Dabei kann jeweils nur der THEN- oder ELSE-Block abgearbeitet werden, daher ist die Rückgabe eindeutig. Diese Funktion können wir in einer externen Datei als eigenes Skript speichern, dann müssten wir aber den Aufruf noch um einen Befehl erweitern, daher wollen wir quasi die Funktion zu Beginn definieren und dann im weiteren Verlauf mittels der bekannten Syntax die Funktion aufrufen. Daher brauchen wir jetzt nur noch folgenden kurzen Befehl:

```
_ergebnis = [wert1, wert2] call fnc_max;
```

Und wir erhalten als Rückgabewert jeweils den größeren der beiden Übergabeparameter. Wir werden euch im Übungsteil noch ein paar Aufgaben zur Vertiefung stellen. Aber auch ohne „magic Variable" könnt ihr sinnvolle Funktionen kreieren, z.B. eine, die euch, wenn ihr sie braucht, die aktuelle Uhrzeit ausgibt:

Listing_36.sqf

```
1  fnc_time =
2  {
3    [WEST, "Base"] sideChat
4      ("Es ist "+str(date select 3)+":"+str(date select 4)+" Uhr.") // 1 Zeile
5  }
```

Die ganze Funktion besteht aus einem einzigen Befehl, nämlich **sideChat**. Dieser kennt zwei verschiedene Syntax-Formen und schreibt im Wesentlichen den übergebenen STRING in den Seitenchat. Die Uhrzeit erhalten wir aus dem ARRAY **date**, das das aktuelle Spieldatum sowie die Uhrzeit beinhaltet. Den Rest solltet ihr dann selbst nachvollziehen. Aufrufen können wir die Funktion mit:

```
call fnc_time;      // Ohne übergebene Parameter
[] call fnc_time;   // Mit einem möglichen Parameterarray, hier leer
```

Magic Variables

Wir wollen – oder vielmehr müssen – uns zum Abschluss des Themas Funktionen noch mit den vielfach erwähnten magischen Variablen befassen. Dabei darf der Begriff „Funktion" in diesem Abschnitt sehr weit aufgefasst werden, denn magische Variablen begegnen uns bei nahezu jedem Skript, bei Schleifen sowie bei Event-Handlern.

Bei magischen Variablen handelt es sich um vordefinierte Bezeichner innerhalb von bestimmten Konstruktionen, die einen bestimmten, festgelegten Zweck und einen festgelegten Gültigkeitsbereich besitzen. Erinnern wir uns an die FOREACH-Schleife. Diese besaß innerhalb des Code-Blockes die magische Variable `_x`. Diese ist in zweifacher Hinsicht festgelegt: Zum einen muss sie so benannt/bezeichnet werden, mit `_i` oder `_n` hätten wir keinen Erfolg. Zum anderen ist ihr Inhalt klar festgelegt, es handelt sich um das aktuelle Element des entsprechenden ARRAYS, über das die FOREACH-Schleife iteriert. Wir merken uns also:

MAGISCHE VARIABLEN WERDEN NICHT VOM BENUTZER ANGELEGT, SONDERN SIND FEST IN SQF IMPLEMENTIERT.

Damit muss man quasi nur wissen, wo einem solche Variablen zur Verfügung stehen und welchem Zweck sie dann dienen. Zum Glück gibt es nur drei wesentliche Einsatzgebiete von magischen Variablen:

1. _x sowie _FOREACHINDEX bei FOREACH und dem erweiterten COUNT
2. _THIS in jedem Skript, jeder Funktion und jedem EventHandler
3. THISLIST in jedem Trigger
4. THIS im Initialisierungsfeld von Objekten im Editor

Da wir 1. schon besprochen haben, können wir uns direkt 2. widmen. Wir hatten diesen Fall ebenfalls bereits im letzten Abschnitt kennengelernt, als wir indirekt Funktionen mit Übergabeparametern gestartet und direkt, als wir unsere eigene Maximum-Funktion umgesetzt haben. Beide Male mussten wir der Funktion ja die Übergabeparameter mitteilen. Dies geschieht generell mit der magischen Variable `_this`. Sie enthält quasi alle Übergabeparameter, die beim Aufruf eines Skriptes oder einer Funktion mitgeteilt worden sind. Damit kann _THIS entweder ein einzelnes Objekt oder ein Array sein und das ist der einzige große Knackpunkt, den man beachten muss.

Schauen wir uns erneut den Aufruf unserer eigenen Maximum-Funktion an:

```
[wert1, wert2] call fnc_max;
```

Offenbar haben wir vor dem Aufruf mittels CALL ein ARRAY mit den entsprechenden Werten angegeben. Diese Werte soll die Funktion nun entsprechend ihres Aufbaus prüfen. Doch wie kommt die Funktion an diese Werte? Vor allem, wenn sie lokal definiert sind, also mit einem Unterstrich beginnen? Nun, indem sie auf die magische Variable `_this` zugreift. `_this` beinhaltet jeweils genau das Objekt, das als Übergabeparameter angegeben wurde. Bei einem einzelnen Wert ist `_this` kein ARRAY sondern ein singuläres Objekt vom entsprechenden Datentyp. Geben wir ein ARRAY an, dann ist `_this` eben genau eine Kopie dieses ARRAYs. A propos Kopie: Man kann sich `_this` also einfach als Kopie des Übergabeparameters vorstellen, exakt das, was vor dem Aufrufbefehl CALL oder SPAWN steht (in diesem Fall, bei anderen Fällen wie Eventhandlern oder Befehlen wie ADDACTION ist es komplizierter).

Wenn nun also `_this` eine Kopie des ARRAYs `[wert1, wert2]` ist, so können wir in gewohnter Weise die Werte mittels SELECT auslesen:

```
_wert1 = _this select 0;
_wert2 = _this select 1;
```

Wir speichern also den erst übergebenen Wert, der ja im ARRAY den Index 0 hat, in eine lokale Variable `_wert1`. Das Gleiche tun wir für den zweiten Übergabeparameter. Damit stehen die zwei übergebenen Zahlen in unserem Beispiel auch innerhalb der Funktion zu Verfügung (wir hatten sie _a und _b genannt). Das war's auch schon. Damit seid ihr in der Lage, eigene Funktionen in vollem Umfang selbst zu schreiben!

Eine Anmerkung noch zur 3. Verwendungsmöglichkeit THISLIST. Diese magische Variable enthält alle Einheiten, die einen Auslöser ausgelöst haben! Sie darf also nicht mit LIST verwechselt werden, was wir ja als ARRAY für alle potentiell auslösenden Einheiten kennengelernt hatten. THISLIST steht nur in Auslösern zur Verfügung und kann sehr gut dafür benutzt werden, zu bestimmen, welche Einheiten die voreingestellte Auslösebedingung erfüllen sollen. Haben wir z.B. einen Auslöser auf einmalig BLUEFOR anwesend gesetzt, so würde er bei jeder Einheit auslösen, die den Trigger-Bereich betritt, insbesondere auch bei Flugzeugen oder KI-Einheiten. Wollen wir dies vermeiden, können wir z.B. ein Flugzeug respektive den Piloten (wir nennen ihn mal `p1`) ausklammern, indem wir in der Bedingungszeile zusätzlich zu `this` (übrigens auch eine magische Variable) Folgendes eingeben:

Damit gilt sowohl die Einstellung des Triggers (`this`) als auch die zusätzliche Bedingung, dass die Einheit `p1` gleichzeitig **nicht** anwesend sein darf (!). Damit kann der Auslöser nicht von der Einheit `p1` ausgelöst werden, egal wie sehr sie sich auch anstrengen mag.

Es sei ergänzend erwähnt, dass Auslöser sehr oft als Startpunkt für Skripte benutzt werden, die mit EXECVM aufgerufen werden. Hier ist es oft nützlich, dem aufgerufenen Skript dann als Übergabeparameter THISLIST mitzugeben.

Variante 4 ist praktisch, um nicht permanent einer Einheit einen Variablennamen geben zu müssen, nur um auf sie zugreifen zu können. Außerhalb des Initialisierungsfeldes ist dies natürlich weiterhin notwendig, aber jede Einheit kann im Editor innerhalb ihres Dialoges über THIS angesteuert werden. Wenn wir also einen beliebigen Code in der Init-Zeile einer Einheit ausführen wollen, genügt THIS als Referenz auf die Einheit:

Listing_37.sqf

```
1  this removeWeapon "M16"; // Entfernt die Waffe M16
2  this setpos [x,y,z];     // Versetzt die Einheit an die Position
3  this setDamage 1;        // Fügt der Einheit den Schaden 1 zu
```

Exkurs – der Aufruf von Funktionen und Skripten im Allgemeinen

Auch wenn wir bereits mehrfach das Ende dieses Kapitels eingeläutet haben – jetzt kommt wirklich das allerletzte Schlusswort, versprochen. Aber wir wollen euch an dieser Stelle auch gar keinen neuen Stoff zumuten, sondern das Gelernte vertiefen und festigen, denn man verliert bei neuen Themen schnell den Überblick, wenn mehr als ein Befehl gleichzeitig vorgestellt wird. Daher wollen wir uns zum Abschluss (wir sagten bereits, dass dieser wirklich final ist) noch einmal die verschiedenen Möglichkeiten des Aufrufes von Skripten anschauen. Da Funktionen nichts anderes als zweckgerichtete Skripte sind, fallen sie selbstverständlich auch unter diesen Abschnitt.

Wir hatten ganz zu Beginn dieses Handbuchs gleich bei unserem ersten Beispiel den Befehl EXECVM kennengelernt. Aus didaktischen Gründen wiederholen wir hier noch einmal den BI Wiki-Eintrag:

Description

Description:	Compile and execute SQF Script. The **optional** argument is passed to the script as local variable _this. Script is compiled every time you use this command. The Script is first searched for in the mission folder, then in the campaign scripts folder and finally in the global scripts folder.

Syntax

Syntax:	Script = argument **execVM** filename
Parameters:	argument: Any Value(s) filename: String
Return Value:	Script - script handle, which can be used to determine (via scriptDone) when the called script has finished.

Examples

Example 1:

```
_handle = player execVM "test.sqf";
waitUntil {scriptDone _Handle};
```

Inzwischen sind wir in der Lage, alle Informationen zu verstehen und auszuwerten – was ein Fortschritt! Bis auf den Hinweis „Compile and execute SQF Script“[38] sollte uns alles bekannt und vertraut vorkommen. Auch hier haben wir, ähnlich wie bereits bei CALL und SPAWN, die Möglichkeit, optionale Parameter anzugeben. Das heißt nichts anderes, als dass wir dem Skript Übergabeparameter mitgeben können, was ja auch bitter nötig ist, wenn das Skript auf irgendetwas reagieren soll. Hier sei nochmal auf die schwammige Unterscheidung zwischen Funktionen und Skripten hingewiesen: Wenn es um den Aufruf geht, gibt es schlichtweg keinen Unterschied. Lediglich wenn wir von Rückgabewerten sprechen, so sind Funktionen strenggenommen immer mit einer Auswertung oder Berechnung verbunden, die wir an einer anderen Stelle gerne weiterverwenden würden. Zurück zu EXECVM: Wir können also Übergabeparameter angeben. Und zwar wie bei CALL und SPAWN in beliebiger Dimension und Größe, also sowohl einen einzigen Parameter als ein ganzes Bündel in Form eines ARRAYs. Auf diese Parameter können wir dann innerhalb des aufgerufenen Skriptes mit der magischen Variable `_this` zugreifen, soweit alles altbekannt. Zudem hat execVM noch eine Besonderheit: Es handelt sich dabei selbst um eine Funktion, die einen Rückgabewert vom Typ Script (Handle) zurückgibt. Dieser lässt sich – wie im Beispiel gezeigt – mit dem Befehl **scriptDone** auswerten, um z.B. die Abarbeitung des Skriptes abzuwarten[39]. EXECVM gibt den Pfad des Skriptes zurück, solange dieses noch ausgeführt wird, und wird in dem Moment zu `<NULL-script>`, indem das Skript abgearbeitet ist. SCRIPTDONE prüft also quasi, ob noch ein Pfad zurückgegeben wird oder eine `Null`-Referenz und gibt nur im letzteren Fall TRUE zurück.

Damit haben wir prinzipiell also vier Möglichkeiten, mit EXECVM ein Skript aufzurufen:

Listing_38.sqf

```
1  execVM "meinSkript.sqf";  //Ohne Rückgabewert und Übergabe-Parameter
2  (param1) execVM "meinSkript.sqf"; //Ohne Rückgabewert und mit 1 Parameter
3  [paramarray] execVM "meinSkript.sqf"; //Ohne Rückgabewert und mit Parameterarray
4  _handle = [...] execVM "meinSkript.sqf"; //Mit Rückgabewert und Parameter
```

Nun ist es leider so, dass diese vier Varianten nicht überall funktionieren. Möchtet ihr ein Skript z.B. aus einem Auslöser heraus starten (in der Aktivierungszeile), so wird Variante 4 zwingend erwartet, d.h. ihr müsst den Rückgabewert in einer Variablen speichern.

[38] Was es mit COMPILE auf sich hat, erklären wir im Kapitel I. 6
[39] Wie man im BI Wiki nachlesen kann, gibt auch der Befehl SPAWN einen Skripthandler zurück.

Fehlermeldung bei Verwendung von EXECVM im Auslöser ohne Rückgabewert

Da im Auslöser natürlich keine lokalen Variablen benutzt werden können, sieht man daher sehr oft folgende Lösung, da man den Rückgabewert gar nicht weiter benutzen möchte:

```
nul = [] execVM "meinSkript.sqf";
```

Man wählt also eine aussagekräftige Variable, von der man weiß, dass sie quasi ein Mülleimer ist. Man sollte nicht auf die Idee kommen, NIL stattdessen zu benutzen, denn NIL ist von SQF als Schlüsselwort reserviert und die Überschreibung wird mit einer entsprechenden Fehlermeldung quittiert.

Innerhalb eines Skriptes könnt ihr aber ein anderes Skript oder eine Funktion mit allen vier vorgestellten Varianten starten, je nachdem, ob und wie viele Übergabeparameter ihr übergeben und ob ihr einen Rückgabewert speichern wollt.

Um die folgenden Unterschiede besser zu verstehen, muss noch gesagt werden, dass EXECVM das aufgerufene Skript in einem neuen Thread startet, was nichts anderes als eine parallele Abarbeitung ist. Wir bewegen uns zum einen in unserem ursprünglichen Skript, wo wir EXECVM benutzen und während EXECVM ein weiteres Skript startet, wird das Ursprüngliche weiter abgearbeitet. Dies sehen wir daran, dass wir z.B. direkt nach dem Aufruf mit EXCEVM eine Variable initialisieren könnten mit `a=3;`. Im aufgerufenen Skript schalten wir einen SLEEP-Befehl `sleep 1;` dazwischen und führen danach `a=1;` aus. Was wird nach dem Ende beider Skripte der Wert von a sein? Richtig, es ist 1 und nicht 3. Eigene Tests ergeben, dass EXECVM das aufgerufene Skript in ca. 0.02-0.03 Sekunden aufgerufen hat. In dieser Zeit können sehr viele Befehle im ursprünglichen Skript ausgeführt werden. Daher ist es wichtig, eventuell auftretende Parallelitäten, die unerwünscht sind, mit `sleep` oder `waitUntil{scriptDone …}` abzufangen.

Nun kommen wir zum Befehl SPAWN. Dieser macht zunächst einmal genau dasselbe wie EXECVM, d.h. er startet das angegebene Script in einem eigenen Thread und führt zu einer parallelen Abarbeitung. Jetzt gibt es nur ein kleines Problem, denn

```
[] spawn "meinSkript.sqf";
```

erzeugt leider eine Fehlermeldung. Offenbar erwartet SPAWN einen Code und keine Zeichenkette:

Jetzt liegt uns aber nur ein Skript vor. Wir wollen natürlich nicht den Inhalt des Skriptes mühsam kopieren und in geschweifte Klammern packen, damit SPAWN seinen Code-Block erhält und zufrieden ist. Wir müssen also – um das Problem zu lösen – das Skript quasi vorkompilieren und SPAWN zur Verfügung stellen. Wir dürfen nicht einen String übergeben, sondern müssen irgendwie an den Code innerhalb des externen Skriptes herankommen. Dafür gibt es drei Lösungen.

Wir hatten oben beim EXECVM-Befehl noch eine Kleinigkeit bisher nicht verstehen können: Was der Hinweis bedeutet, dass EXECVM ein Skript kompiliert und ausführt. Jetzt wissen wir es: EXECVM tut genau den Zwischenschritt, den SPAWN nicht macht: es kompiliert die angegebene Skript-Datei, was so viel bedeutet wie eine Übersetzung in ausführbaren Code. Zusätzlich führt es diesen dann auch gleich aus.

Daher suchen wir nach entsprechenden Befehlen, die uns den Zwischenritt des Kompilierens liefern. Dabei stoßen wir zunächst auf **compile**. Dieser Befehl ist wunderbar klein, einfach und liefert augenscheinlich das, was wir suchen: Er wandelt einen Ausdruck in einen Code um. Da SPAWN einen Code-Block erwartet, ist der Rückgabewert schon einmal der richtige. Jetzt gibt es aber ein Problem mit dem Argument „expression“. Dieses muss vom Typ STRING sein und einen korrekten SQF-Befehl beinhalten. Wir haben aber

lediglich den Dateinamen als STRING. Daher wird `meinSkript.sqf` kaum ein ausführbarer SQF-Code sein. Wir brauchen immer noch den Inhalt des Skriptes, den wir dann mit COMPILE umsetzen können.

Zur Lösung des Problems finden wir den Befehl **loadFile**. Dieser gibt als Rückgabewert den Inhalt einer Datei im Datentyp STRING wieder. Perfekt. Wir fassen die gewonnen Kenntnisse als kleines Beispiel zusammen:

Listing_39.sqf

```
1  _inhalt = loadFile "meinSkript.sqf"; // Lädt den Inhalt und speichert ihn
2  _code = compile _inhalt; // Erzeugt aus dem Inhalt ausführbaren Code
3  [] spawn _code // Ruft das Skript per spawn-Befehl auf, extra Thread!
```

Alternativ natürlich:

```
[] spawn (compile loadFile "meinSkrip.sqf")
```

Damit haben wir mehr oder weniger den Befehl EXECVM nachgebaut, denn das Ergebnis ist jetzt identisch. Ein Hinweis: Kommentare scheinen beim Befehl COMPILE einen Fehler zu verursachen. Um dennoch z.B. nicht alle Skripte abzuändern, könnt ihr den Befehl **comment** benutzen.[40]

Jetzt wird sich wahrscheinlich jeder fragen, warum man um alles in der Welt einen so umständlichen Befehl wie SPAWN benutzen sollte, wenn es doch EXECVM gibt. Die Antwort liegt wie immer im Argument. Auch wenn das Ergebnis beider Befehle identisch aussehen mag, so kann SPAWN doch wesentlich vielseitiger eingesetzt werden. Denn EXECVM erwartet zwangsweise eine Skriptdatei als Argument. Was ist aber, wenn ihr gar kein Skript vorgeschrieben habt, sondern einfach nur einen bestimmten, eventuell während der Mission erst generierten Code ausführen müsst? Dann braucht ihr SPAWN, denn damit könnt ihr jeden Code als Skript ablaufen lassen, da ihr ja jeden Code als STRING schreiben und mit COMPILE in Code umwandeln könnt.

SPAWN ist vor allem dann sinnvoll, wenn ihr Code ausführen wollt, der euer aktuelles Skript aufhalten bzw. pausieren würde. So könnte es ja sein, dass ihr eine while-Schleife braucht, um einen Befehl nur solange auszuführen, wie eine bestimmte Einheit am Leben ist. Dies würde in eurem Hauptskript aber solange zu einer Pausierung der Abarbeitung führen, bis die while-Schleife beendet ist. Da es bei kurzen Skripten keineswegs immer sinnvoll ist, dafür eigene Skripte oder Funktionen zu schreiben, kann der Code auch einfach mittels SPAWN-Befehl ausgelagert werden.

Wir haben Lösung eins sehr ausführlich beschrieben und möchten jetzt nur noch zwei alternative Lösungswege für das Kompilieren vorstellen, die in der Praxis eher Verwendung finden: **preprocessFile** und **preprocessFileLineNumbers**. Beide tun prinzipiell genau dasselbe wie LOADFILE, d.h. sie speichern den Inhalt einer sqf-Datei in einen STRING. Der Grund, warum man diese beiden dem letzteren vorzieht, liegt in der Fehlerverarbeitung. PREPROCESSFILELINENUMBERS bietet die beste Fehlerauswertung und unterstützt mehr Befehle, so z.B. die oben noch problematischen Kommentare mit `//`. Man kann sich das so vorstellen, dass LOADFILE die Datei ohne jede Bearbeitung einfach nur öffnet und so speichert, wie es die Datei vorfindet, während PREPROCESSFILELINENUMBERS bereits einige Dinge vorsorglich vornimmt – unter anderem schmeißt es direkt alle Kommentare raus. Ihr könnt beide Ergebnisse mit der ACE-Debug-Konsole vergleichen! Ohne zu sehr ins Detail zu gehen, raten wir zum Gebrauch von PREPROCESSFILELINENUMBERS, da Fehlerangaben mit Zeilennummern ausgegeben werden.

Jetzt wissen wir das Wichtigste in Bezug auf SPAWN und EXECVM, bleibt noch CALL. Das geht recht schnell: CALL ist identisch mit SPAWN, nur wird der Code im aktuellen Skript ausgeführt, also kein neuer Thread gestartet. Damit kann vereinfacht gesagt werden: Habt ihr kurzen, einfachen Code, der nur einmal ausgeführt werden muss, so benutzt CALL. Habt ihr längeren, größeren Code oder Code, der das Skript pausiert, weil er z.B. auf eine Bedingung wartet, nehmt SPAWN. Das war's.

---

[40] Wir benutzen ab jetzt [] immer, um anzudeuten, dass beliebige Parameter beim Aufruf angegeben werden können. Wir haben ja bereits gelernt, dass wir jedes Skript mit Übergabeparametern starten dürfen. Zugriff im Skript mittels `_this`.

Als aller aller aller Letztes weißen wir euch noch auf die mächtige Möglichkeit hin, Funktionen und Skripte, die öfters Verwendung finden, in Variablen vorrätig zu halten. Im Grunde haben wir dies bereits oben im ausführlichen Beispiel zu SPAWN benutzt. Wenn wir ein Skript oder eine Funktion oder einen Code in STRING-Format mit `compile preprocessFileLineNumbers` auslesen, so erzeugen wir einen STRING, der fertigen Code enthält. Diesen brauchen wir nur noch an geeigneter Stelle mittels CALL oder SPAWN aufzurufen. Dabei ist es extrem praktisch, dass wir dies in jedem unsere Skripte tun können, wenn wir die Variable, die den Code enthält, global definieren. Wir erinnern uns an unsere Maximum-Funktion. Diese speichern wir jetzt in eine externe Datei mit z.B. dem Namen „fnc_max.sqf“. Damit ist klar, dass es sich um eine Funktion handelt. In der INIT.SQF legen wir dann einmal den Inhalt als global bekannt fest:

```
fnc_max = compile preprocessFileLineNumbers "fnc_max.sqf";
```

Wir brauchen auch nicht innerhalb der Datei „fnc_max.sqf“ die Funktion selbst noch – wie oben beschrieben – mit `fnc_max = {…}` aufsetzen, denn dies tun wir ja jetzt in der INIT.SQF. Eine Funktion müssen wir nur dort mit Bezeichner und geschweiften Klammern definieren, wo wir sie auch in derselben Datei verwenden wollen. Alternativ kann man aber die Datei auch mit `fnc_max = {…}` belassen und ruft dann in der INIT.SQF mittels `execVM „fnc_max.sqf“` die Datei nur noch auf. Durch den Aufruf wird die Datei ausgeführt und die globale Variable `fnc_max` angelegt. Beide Varianten scheinen mir gleich gut. Der Aufruf der Funktion kann dann gewohnt mit

```
_ergebnis = [wert1, wert2] call fnc_max;
```

erfolgen und das Ergebnis gespeichert werden. Der große Vorteil ist jetzt, dass wir diesen Aufruf in jedem Skript benutzen können. Da `fnc_max` als globale Variable definiert wurde, ist sie von Beginn an beim Spieler bekannt. Daher brauchen wir nie wieder die Funktion selbst mit COMPILE zu kompilieren oder die Datei mit EXECVM neu zu laden, wir haben sie einmal kompiliert und sparen bei jedem weiteren Aufruf bare Rechenzeit.

Übungen

[1] Umgang mit dem Funktions-Viewer üben: Findet eine Funktion, die es ermöglicht, dem Spieler Ausrüstung hinzuzufügen. Fügt ihm dann ein Fernglas sowie einen M136-Launcher hinzu.

Hinweis: Im englischen heißt Inventar inventory. Der Klassennamen für das Fernglas ist „Binocular“, für den M136 unter ACE „ACE_M136_CSRS“.

[2] Für diese Übung müsst ihr das nachfolgende Beispielprojekt gut verstanden und am besten selbst nachvollzogen und geschrieben haben. Wir wollen uns jetzt um ein wenig Feintuning kümmern. Zum einen wäre es sehr günstig, wenn die Funktion neben der freien Sitzplatzanzahl auch die Distanz zwischen Fahrzeug und Spieler zurückgeben würde, denn wir wollen dem Spieler ja auch verraten, wo das Fahrzeug zu finden ist (und ob sich der Weg lohnt). Außerdem haben wir gesehen, dass das Skript im Moment noch Feindfahrzeuge anzeigt. Eure Aufgabe besteht also darin, einen zusätzlichen Rückgabewert für die Entfernung hinzuzufügen und die Abfrage der freien Fahrzeuge im Hauptskript auf West- oder Zivilfahrzeuge zu beschränken.

I. 7. BEISPIELPROJEKT

Wir wollen jetzt ein komplexes Beispiel bzw. eine komplexe Übung anbieten, damit ihr das Gelernte im Zusammenhang begreifen und Anwenden könnt. Wir wollen uns eine eigene Funktion schreiben und diese als externes Skript auslagern. Diese Funktion wollen wir dann in einem Beispielprogramm exemplarisch verwenden. Die Funktion soll ein übergebenes Fahrzeug auf freie Plätze überprüfen. Als Rückgabewert soll sie die Anzahl der freien Plätze zurückgeben.

Unser Hauptskript hat die Aufgabe, dem Spieler mitzuteilen, ob aktuell Fahrzeuge auf der Karte vorhanden sind, in die er einsteigen kann.

Das ganze Projekt ist etwas fortgeschritten und soll euch die Möglichkeiten von ArmA aufzeigen. Daher haben wir uns entschlossen, das ganze zwar als Übung aufzufassen, aber hier nicht zehn Hinweise zugeben und dann anschließend im Lösungsteil nochmal fünf Seiten zu schreiben. Wir werden daher das gesamte Projekt gemeinsam mit euch an dieser Stelle entwickeln und die Lösung im selben Zuge präsentieren.

Nötige Befehle:

Zunächst brauchen wir einen Befehl für die aktuellen Passagiere. Dies können wir mit **assignedCargo** lösen.

Für die Abfrage der Gesamtplätze müssen wir in das Auslesen von CONFIG-Dateien einsteigen. Wir brauchen den bekannten Befehl TYPEOF sowie **getNumber**.

Um sicherzustellen, dass das Skript nicht für falsche Anfragen missbraucht wird, nutzen wir **isKindOf**.

Außerdem brauchen wir den berühmt-berüchtigten Befehl zum vorzeitigen Beenden: **exitWith**. Damit ist unsere Funktion dann auch zur Gänze erschlagen.

Dann wollen wir uns an die Arbeit machen. Wir beginnen mit der Funktion.

Listing_40.sqf

```sqf
/* ----------------------------------------------------------------------------
Function: fnc_freiePlaetze

Description:
  Gibt für ein Fahrzeug die Anzahl der freien Passagierplätze ohne Fahrer,
  Schütze, Kommandant aus

Parameters:
  _vehicle - Fahrzeug, das geprüft werden soll

Returns:
  Number

Example:
  (begin example)
    (auto) call fnc_freiePlaetze;
  (end)

Author:
  [3.JgKp]James, (c) 2013
---------------------------------------------------------------------------- */
private ["_vehicle", "_class", "_passagiere", "_maxAnzahl", "_aktuelleAnzahl"];

_vehicle = _this;

// Auslesen des Klassennamens
_class = typeOf _vehicle;

//Sicherheitsabfrage
if (!(_class iskindof "Car") && !(_class iskindof "Tank")) exitWith {false};

// assignedCargo gibt ein Array mit allen Passagieren ohne Fahrer, Schütze,
// Kommandant zurück
_passagiere = count assignedCargo _vehicle;

// Auslesen des Wertes transportsoldier aus der config des Fahrzeugs
_maxAnzahl = getNumber (configFile>>"CfgVehicles">>_class>>"transportsoldier");

_aktuelleAnzahl = _maxAnzahl - _passagiere;

// Rückgabewert
_aktuelleAnzahl
```

Schön, oder? Der Kopf ist eine Art Meta-Datei und sollte so ähnlich bei allen Funktionen und Skripten aussehen, die ihr eventuell an andere weiterreicht. Da unsere Funktion jeweils nur ein Fahrzeug überprüfen soll, reicht ein Übergabeparameter. Entweder ist dies der Variablennamen des Fahrzeugs (z.B. `auto`) oder sein Objektname, den die Spielengine uns liefern wird. In der ersten Anweisung der Funktion (Zeile 25) weisen wir den Übergabeparameter einer lokalen Variablen zu. Als nächsten brauchen wir den Klassennamen der Datei, um auf die CONFIG zugreifen zu können. Dies können wir mit dem Befehl TYPEOF abfragen und in eine lokale Variable `_class` speichern. Jetzt führen wir eine erste Sicherheitsabfrage ein: Obwohl wir im Hauptskript nur korrekte Übergabeparameter übergeben werden, können wir nicht sicherstellen, dass unsere Funktion nicht zu einem anderen Zeitpunkt falsch aufgerufen und zweckentfremdet wird, was einen Fehler zur Folge hätte. Denn nur Fahrzeuge werden wahrscheinlich über einen Wert für Passagiere verfügen, von Objekten wie einem Haus oder dem Spieler erwarten wir hingegen einen Fehler, wenn wir diese der Funktion übergeben. Daher schließen wir mit der Abfrage in Zeile 31 zwei Fälle aus:

Entweder das übergebene Objekt gehört zur Kategorie „Car“ oder zur Kategorie „Tank“. Dies prüfen wir mit ISKINDOF. Der Befehl vergleicht den linken Klassennamen mit dem rechten. Da wir diese Bedingung verneinen (!), heißt dies: Wenn das Objekt NICHT vom Typ „Car“ ist UND NICHT vom Typ „Tank“, mache Folgendes….Was soll das Skript in diesem Falle tun? Am besten, sich beenden. Dann haben wir keine weiteren Probleme. Wenn also jemand ein falsches Objekt übergibt, macht die Funktion direkt dicht. Das bewirkt man mit dem Befehl EXITWITH. Wir werden im Kapitel „Vorzeitiges Beenden von Schleifen und Skripten - Sprungstellen“ näher auf diese und andere Möglichkeiten des vorzeitigen Beendens eingehen. Wichtig ist nur an dieser Stelle: EXITWITH beendet das Skript, gibt aber die Möglichkeit, dabei noch einen Code auszuführen. Wir erinnern uns: Eine Funktion hat immer einen Rückgabewert. Wenn die Funktion beendet wird, müssen wir uns also überlegen, was in diesem Falle zurückgegeben werden soll. Wir haben prinzipiell zwei Möglichkeiten: Entweder wir codieren den falschen Aufruf durch eine Zahl oder lassen einen BOOLEAN zurückgeben. In unserem Beispiel haben wir uns für einen Wahrheitswert entschieden. Da ein Rückgabewert einer Funktion immer eine Anweisung ist, reicht es, FALSE als Anweisung zu schreiben, wir brauchen keinen weiteren Befehl (wie RETURN in Java/C#).[41] Wir fassen zusammen:

Listing_41.sqf

```
1  (auto) call fnc_freiePlaetze; // Korrekter Aufruf. Gibt x ϵ [0, n] zurück
2  (player) call fnc_freiePlaetze; // Falscher Aufruf. Gibt false zurück
```

Wenn das Skript über diese Stelle hinaus ist, heißt das, es wurde nicht vorzeitig beendet und wir können mit der eigentlichen Arbeit beginnen. Wir prüfen jetzt die tatsächlich freien Plätze unseres Fahrzeugs mit ASSIGNEDCARGO. Der Befehl gibt ausschließlich ein Array mit Einheiten zurück, die auf den Passagierplätzen zugewiesen wurden, also sind Fahrer, Schütze und Kommandant von vornherein nicht enthalten. Mit COUNT zählen wir einfach die Anzahl der Elemente. Damit erhalten wir bei einem völlig leeren Fahrzeug (bezogen auf die Passagiere!) oder bei einem Fahrzeug ohne Passagiersitze eine 0, ansonsten eine Zahl zwischen 1 und n.

Nun kommen wir zu Zeile 39. Zunächst wollen wir überlegen, was wir brauchen: Wir wollen ja die momentan freien Plätze in einem Fahrzeug ermitteln (zumindest bezogen auf die Passagierplätze, ansonsten würde das Skript noch umfangreicher). Dazu wollen wir die maximal freien Plätze von den aktuell schon besetzten abziehen. Die aktuell besetzten haben wir eben ermittelt, was uns fehlt sind die insgesamt verfügbaren. Diese müssen ja aber irgendwo abgreifbar sein! Immerhin weiß das Fahrzeug ja, wann es voll ist. Da jedes Fahrzeug seine eigenen Sitzplatzkapazitäten hat, ist anzunehmen, dass wir diese Information irgendwo aus dem Fahrzeug selbst auslesen können. Leider existiert nur hierfür kein Befehl! Daher müssen wir einen Umweg über die fahrzeugeigene CONFIG gehen. Eine CONFIG-Datei ist so etwas wie der Bauplan eines jeden Objektes. Sie legt genau fest, wie das Objekt aufgebaut, texturiert und animiert ist, über welche Attribute es verfügt und wie es sich im Spiel verhält.

Schauen wir uns einmal eine mögliche Fahrzeug-CONFIG an: `class Ural_Base_withTurret: Truck`. Wir werden im Teil Teil II – SQF im Multiplayer ausführlich auf den SIX Config Browser eingehen. Er zeigt euch quasi alle Gegenstände in Arma2 OA und ACE mit Klassennamen und zugehörigen Subklassen/Elternklassen sowie Magazinen (bei Waffen). Man findet solch eine Config am leichtesten, indem man bei einem Referenzfahrzeug nachschaut. Dazu wählt man im SIX Config Browser z.B. `Vehicles >> Classlist`. Dabei sollte man darauf achten, dass die korrekte Version unter CfgVehicles eingestellt ist, also ACE with ACEX: 1.14 RC1 b585 (zum Zeitpunkt dieses Handbuches). Dann kann man mit der Combobox `vehicleclass` direkt nach Klassen sortieren, in unserem Fall „Car“ als Oberklasse. Jetzt kann man ein beliebiges Fahrzeug auswählen.[42] Jetzt erscheint eine detaillierte Übersicht des angewählten Fahrzeuges. Wichtig für uns ist der blau hervorgehobene Link CONFIG. Dabei dürfte das Ergebnis sehr spartanisch ausfallen. Das liegt daran, dass fast alle Fahrzeuge sogenannte Kindklassen (vererbte Klassen)

[41] Genauso möglich und eventuell sogar besser für weitere Anwendungen wäre eine numerische Rückgabe in Form von -1.
[42] In der Übersicht erfahrt ihr bereits den Klassennamen, den ihr z.B. zum Spawnen braucht. Außerdem seht ihr Bewaffnung und mögliche Magazine dafür.

von Elternklassen sind. Dies erkennt man daran, dass hinter dem Klassennamen eine weitere Klasse - durch „:“ separiert - angegeben ist. So ist im obigen verlinkten Beispiel also der `Ural_Base_withTurret` eine Kindklasse von Truck. Zum Glück kann man sich wunderbar im Config Browser durch die Ebenen bewegen, indem wir mit einem Linksklick auf die Elternklasse jeweils eine Ebene aufsteigen. Wir haben das für euch im obigen Beispiel bereits getan, so dass ihr nicht auf Truck klicken braucht. Wir suchen nämlich immer noch die Anzahl an Passagiersitzplätzen. Dazu bewegen wir uns prinzipiell solange von Elternklasse zu Elternklasse, bis wir eine finden, die zum ersten Mal das gesuchte Attribut enthält! Gehen wir dabei zu weit nach oben, könnten wir einen veralteten Wert auslesen, weil die vererbte Klasse die Eigenschaft eventuell mit einem anderen Wert überschreibt! Daher ist es wichtig, bei der untersten Ebene zu beginnen. Wir ihr dem obigen Link entnehmen könnt, kann eine CONFIG unglaublich umfangreich sein! Wir ersparen euch daher das lange Gesuche und verraten euch den Namen des gesuchten Attributes: **`transportsoldier`** `= 14`. Auch hierfür gibt es im BI Wiki eine gute Seite. Damit kennen wir das Attribut, nach dem wir suchen müssen. Zurück in unsere Funktion und zu Zeile 39:

Was wir jetzt noch brauchen, ist der Wert, den das übergebene Fahrzeug für das Attribut `transportsoldier` in der CONFIG eingetragen hat. Dies geschieht, indem wir uns quasi von der obersten CONFIG nach unten bewegen. Dazu beginnen wir im root-Verzeichnis mit dem Befehl **`configFile`**. Für ein besseres Verständnis zeigen wir euch mal den Aufbau der CfgVehicles. Hieran kann man sehr gut die Vererbungshierarchie verfolgen. Die oberste Ebene aller Fahrzeuge ist die CfgVehicles (oben links). CfgVehicles umfasst dabei sehr viele Objekte, unter anderem auch Einheiten und Tiere. Da der Befehl TYPEOF die Klasse zurückliefert, können wir tatsächlich direkt auf dieses Objekt über die Hierarchie zugreifen. Dafür müssen wir nur noch wissen, dass man sich zwischen den Hierarchien nicht wie bei Pfaden mit „/“ beweget, sondern mit Doppelpfeilen „>>“. Das bedeutet, wir erreichen unser Attribut über folgende Angabe:

```
configFile >> "CfgVehicles" >> _class >> "transportsoldier";
```

Damit lesen wir aber bisher nur den Pfad zu dieser Datei bzw. dem Attribut aus. Da wir aber den Wert brauchen, setzen wir das Ganze als Argument von GETNUMBER. Damit haben wir dann auch endlich die gesuchte Kapazität an Passagieren.

Der Rest der Funktion ist ohne Weites verständlich: Wir bilden als vorletztes noch die Differenz zwischen der Gesamtkapazität und der aktuell belegten Anzahl an Plätzen und geben schlussendlich diesen Wert als Rückgabewert zurück. Die Funktion speichern wir in einer SQF-Datei mit dem Namen „fnc_freiePlaetze.sqf“ im Ordner „funktion“. Danach können wir in der INIT.SQF die Funktion in einer globalen Variablen speichern:

Listing_42.sqf

```
1  fnc_freiePlaetze = compile preprocessFileLineNumbers
   "funktion\fnc_freiePlaetze.sqf"; // 1 Zeile
```

Dabei handelt es sich um eine Zeile ohne Umbruch. Damit steht fortan in allen Skripten diese Funktion zur Verfügung. Gehen wir über zu unserem Hauptskript:

Listing_43.sqf

```
1  // Skript, das dem Spieler mögliche freie Fahrzeuge nennt
2
3  // zunächst bestimmen wir alle Fahrzeuge auf der Karte
4  _fahrzeugListe = vehicles - ["ACE_JerryCan_15"];
5
6  // Schleife über alle Fahrzeuge
7  {
8    // Wenn Fahrzeugplätze = 0 -> Löschen aus Liste
9    if ((_x) call fnc_freiePlaetze == 0) then
10   {
11     _fahrzeugListe = _fahrzeugListe - [_x]
12   }
13 } forEach _fahrzeugListe;
14
15 //Ausgabe an den Spieler
16 hint ("Aktuell verfügen folgende Fahrzeuge über freie Plaetze: \n "+
   str(_fahrzeugListe))
```

Wie wir sehen, fällt das Skript bereits deutlich kürzer aus als die eigentliche Funktion, ein Hauptanliegen beim Einführen von Funktionen. Außerdem ist der Code wesentlich besser lesbar, einfacher zu warten und – wie oft erwähnt – der Aufruf der Funktion ist ressourcenschonender und schneller. Wir beginnen in Zeile 4 mit dem Auslesen aller auf der Karte verfügbaren Fahrzeuge. Dabei haben wir das Problem, dass der Befehl VEHICLES auch ACE-Benzinkanister umfasst. Daher schmeißen wir diese kurzerhand aus dem ARRAY und nennen das bereinigte ARRAY `_fahrzeugListe`. Dies geht so schön einfach, weil der „–“-Operator alle Elemente und nicht etwa nur das letzte löscht. Jetzt wollen wir für jedes Fahrzeug aus `_fahrzeugListe` prüfen, ob es freie Plätze für Passagiere hat. Genau dafür haben wir ja unsere Funktion fnc_freiePlaetze geschrieben. Also nutzen wir eine FOREACH-Schleife, iterieren über alle Fahrzeuge und prüfen, ob die freien Plätze gleich 0 sind. Wenn dies so ist, ist das Fahrzeug für uns nicht von Interesse und wir können es auf dem ARRAY löschen. Da wir es später dem Spieler als Meldung anzeigen wollen, können wir nicht mit dem SET-Befehl und NIL arbeiten, sondern müssen das ARRAY echt verkürzen.

Das war es im Grunde. Als letztes geben wir das ARRAY ohne große Umstände einfach als solches aus, indem wir es in einen String mit dem Befehl STR konvertieren. Dabei bewirkt der Befehl \n einen Umbruch. Als letztes speichern wir unser Hauptskript in der Datei „freieFahrzeuge.sqf“. Um das Skript im Spiel zu testen, können wir es mittels ACE-Debug-Konsole einfach aufrufen (`execVM „freieFahrzeuge.sqf“`) oder aber einen (Funk)Auslöser dafür basteln. Die Ausgabe sieht dann wie folgt aus:

In meinem Beispiel habe ich einen BW LKW sowie einen Marder mit Namen benannt, einen VW Golf und ein Feindfahrzeug aber so auf der Karte platziert. Entsprechend gibt die Funktion alle 4 Fahrzeuge als Treffer aus. Der Panzer sowie der Traktor werden nicht angezeigt, da sie keine freien Passagierplätze haben. Die Ausgabe ist nicht sonderlich schön, aber es funktioniert! Was eine Leistung – finden wir wenigstens. Um das Verfeinern kann man sich im Anschluss kümmern. Siehe dazu Übung [2] dieses Kapitels.

Wir hoffen, dass euch dieses Beispielprojekt etwas gefordert und vor allem Spaß gemacht hat. Einiges ist bestimmt noch unsicher oder unklar, aber wir hoffen, dass wir in diesem Projekt viele bisher besprochene Konzepte sinnvoll vereinigen konnten. Es würde uns jedenfalls freuen, wenn es euch geholfen hat.

I. 8. EVENT-HANDLER

Wir nähern uns dem Ende des ersten Teils. Das bedeutet, wir haben fast nichts mehr an Grundlagen zu vermitteln, ehe wir uns in die Teile II und III begeben können. Naja, fast ist eben nur fast .

Was wir euch in diesem Kapitel präsentieren wollen, ist die vielleicht mächtigste der bisher erlernten Möglichkeiten von SQF. Bei Funktionen haben wir bereits einen kurzen Blick auf bereitgestellte Funktionalitäten geworfen, auf die wir anderweitig gar keinen Zugriff hätten bzw. deren Umsetzung für uns selbst nur mit sehr viel Aufwand möglich wäre. Jetzt lernen wir eine neue Klasse von – nennen wir es ruhig Befehlen – kennen, die überhaupt nicht selbst zu programmieren sind! Es gibt schlichtweg keine Möglichkeit, die jetzt vorgestellten Funktionalitäten anderweitig umzusetzen, daher wollen wir sie euch natürlich umso mehr nahebringen.

Wir reden von **Event-Handlern**. Ein Event-Handler (EH) ist an sich schnell erklärt: Es handelt sich dabei quasi um einen Prozess, der permanent überwacht, ob ein bestimmtes Ereignis eintritt. Wenn dies der Fall ist, führt er nutzerspezifischen Code aus. Mit anderen Worten: EH ermöglicht das Ausführen von Skripten nach bestimmten Ingame-Ereignissen. Das Konzept ist jedem vertraut, der moderne Programmiersprachen kennt (Java, C#: Event-Listener). Das wirklich geniale an EH ist nun, dass ArmA2 (und natürlich auch 3) eine große Vielfalt an EH bietet, die fast alle möglichen Ereignisse abdecken.

Was kann man sich nun unter so einem Ereignis vorstellen? Einige Beispiele: Das Abfeuern der Primärwaffe, das Erleiden von Schaden oder das Getötet-werden, das Einsteigen oder Aussteigen aus Fahrzeugen, das Ausführen einer Animation. Dabei können EH sowohl an Einheiten als auch an Objekten „befestigt“ werden.

Wir sprechen im Allgemeinen davon, einem Objekt einen EH hinzuzufügen oder diesen wieder zu entfernen, wenn wir damit meinen, dass wir fortan das Objekt auf ein bestimmtes Ereignis hin überwachen. Um einen EH hinzuzufügen, benötigen wir den Befehl **addEventHandler**.

Betrachten wir uns zum Einstieg die Seite des BI Wiki. Wir erfahren zunächst den sehr wichtigen Hinweis, dass eine Einheit eine beliebige Anzahl an EH haben kann. Vor allem kann sogar derselbe Typ EH mehrfach hinzugefügt werden. Des Weiteren können wir einen EH natürlich auch wieder mittels **removeEventHandler** entfernen. EH werden durch Indizes sortiert, d.h. der erste hinzugefügte EH erhält den Index 0, der nächste dann 1 usw. Damit können EH entsprechend ihres Index‘ gezielt wieder entfernt werden.

Description

Description: Adds an event handler to a given unit. For more information about event handlers and their types check the scripting topic Event handlers in this reference. You may add as many event handlers of any type as you like to every unit, if you add an event handler of type "killed" and there already exists one, the old one doesn't get overwritten. Use removeEventHandler to delete event handlers.

Every event will create an array named **_this**, which contains specific information about the particular event. (e.g. the "killed" EH will return an array with 2 elements: the killed unit, and the killer.)

Syntax

Syntax: Number = object **addEventHandler** [type, command]

Parameters: object: Object

[type, command]: Array

type: String - Event Handler type

command: Code or String - code that should be executed once the event occurs

Return Value: Number - The index of the currently added event handler is returned. Indices start at 0 for each unit and increment with each added event handler. (optional)

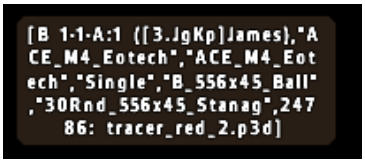
Der nächste wichtige Hinweis betrifft die magische Variable _THIS. Wir erinnern uns, dass magische Variablen hauptsächlich bei Schleifen, Skript- oder Funktionsaufrufen sowie eben bei Event-Handlern vorkommen. Hier übernehmen sie nun eine sehr wichtige und für uns zunächst neue Funktion: Sie beinhalten zwar wie gewohnt die Übergabeparameter, liefern aber im Falle von EH zusätzliche Informationen, die wir als Nutzer gar nicht selbst angegeben haben. Wenn wir an Funktionen denken, so beinhaltet _this immer

genau die Argumente, die wir vor execVM, spawn oder call angegeben haben. Bei Event-Handlern ist dies anders: _THIS wird vom EH jeweils selbst gefüllt, wir als Skripter geben gar keine Übergabeparameter an, vielmehr teilt der EH dem Code-Block alle nötigen Informationen selbst mit. Daher ist im Hinweis auch klar zu lesen, dass der EH selbst die Variable _THIS erzeugt. Um das alles besser zu verstehen, machen wir einen kleinen Test:

Listing_44.sqf

```
1   _number = player addEventHandler ["Fired", {hint str(_this)}];
```

Wir betreiben mit diesem Einzeiler noch keine Magie: Zunächst fügen wir dem Spieler einen neuen EH „Fired“ hinzu, der bei jedem Abschießen einer Waffe ausgelöst wird.[43] Gemäß der Beschreibung des Befehls kommt an zweiter Stelle der auszuführende Code, hier in der Variante mit geschweiften Klammern, um den Datentyp Code klar zu symbolisieren. Dieser wird also immer dann ausgeführt, wenn der EH „feuert“, womit man allgemein auslöst meint. Immer wenn der Spieler also eine Waffe abfeuert, wird `hint str(_this)` ausgelöst. Da der EH selbst die Variable _THIS erzeugt, geben wir also lediglich jedes Mal aus, was der EH eigentlich an Übergabeparametern für den Code-Block so alles zu bieten hat. Ergebnis:



Wir finden eine genaue Beschreibung der Übergabeparameter natürlich im BI Wiki. Der Reihe nach sind dies: die feuernde Einheit (ich), die Waffe, die Mündung, der Feuermodus (Einzel, Salve, Vollautomatik), die Munitionsart, der Klassennamen des Magazins und das Projektil. Jeder EH übergibt also mit _THIS genau das, was für sein spezifisches Ereignis von Interesse sein könnte, mal mehr und mal weniger ausführlich. Wir können jetzt jede erdenkliche Skriptidee im Code-Block umsetzen, vor allem natürlich auch ein Skript aufrufen. Da wir in diesem Skript dann auf die übermittelten Daten zugreifen wollen, müssen wir _THIS mitübergeben:

```
player addEventHandler ["Fired", {(_this) execVM "meinSkript.sqf"}];
```

Damit sind wir in vertrautem Terrain, sollten aber genau auf die Klammern achten! Wir übergeben ja bereits ein ARRAY, denn _THIS wird bereits vom EH mit mindestens zwei Elementen angelegt. Da wir dieses ARRAY an unser Skript übergeben wollen, reichen runde Klammern aus, da _THIS **ein** Argument ist, aber vom Datentyp ARRAY. Würden wir stattdessen zum zunächst plausibel erscheinenden `[_this]` greifen, hätten wir ein kleines Problem, da wir soeben ein zweidimensionales ARRAY angelegt haben und folglich im Skript mit der Zeile

```
_param = _this; // Speichern der Übergabeparameter in einer lokalen Variable
```

nicht das gewünschte Ergebnis erhielten. Denn _THIS ist ein zweidimensionales ARRAY, daher müssten wir

```
_param = _this select 0;
```

schreiben. `_param = _this` führt aber mit runden Klammern im Aufruf zum Erfolg.

Damit stehen uns alle oben genannten Übergabewerte im Skript zur Verfügung, wir müssen lediglich daran denken, dass `_param` selbst ein ARRAY ist, daher spart man sich oft diesen Schritt und speichert die Übergabewerte direkt in den Variablen, die man auch weiter verwenden möchte. Ich rate dazu, immer in den Kopf des Skriptes aus dem BI Wiki alle zur Verfügung stehenden Argumente zu kopieren:

[43] Oben war die Rede von Primärwaffe. Dazu ist leider etwas mehr Aufwand notwendig, aber nicht viel.

Listing_45.sqf

```
1  // Passed array: [unit, weapon, muzzle, mode, ammo, magazine, projectile]
2  _unit = _this select 0;
3  _weapon = _this select 1;
4  _ammo = _this select 4;
5  _magazine = _this select 5;
6
7  if (_weapon == primaryWeapon player) then
8  { hint "Es wurde die Primärwaffe abgefeuert" }
```

Wir haben in den Zeilen 2 bis 5 alle für uns relevanten Übergabewerte ausgelesen. Da wir nicht alle benötigen, müssen wir auf den korrekten Index hinter SELECT achten. Danach kommt unser eigentliches Skript: Wir wollen prüfen, ob der Spieler seine Primärwaffe abgefeuert hat. Dazu steht uns der Befehl **primaryWeapon** zur Verfügung. Auf diese Weise könnten wir über die Magazinklasse auch prüfen, ob eine schallgedämpfte Waffe abgefeuert wurde. Wunderbare Übungsaufgabe!

Damit sind wir am Ende dieses kurzen Kapitels. Wie ihr seht: Mit den bisherigen Konzepten können neue immer schneller angeeignet werden. Wir haben Event-Handler als neue Befehle kennengelernt, die in der Lage sind, selbstdefinierten Code oder eigene Skripte genau dann zu starten, wenn bestimmte Ereignisse eintreten. Da wir aber den Aufruf von Skripten sowie magische Variablen bereits kennengelernt haben, waren wir bestens gerüstete. Alles, was wir neu lernen mussten, ist die Syntax des Befehles selber sowie die Funktionsweise.

Apropos Funktionsweise: Der Aufruf und die Abarbeitung des obigen Beispielskriptes dauert ca. 0.02-0.06 Sekunden. Die direkte Ausführung des Codes im EH selbst dauert 0 Sekunden. Mit anderen Worten: Der EH selbst führt den Code natürlich ohne jede Verzögerung aus, die Zeit ist nicht messbar, also kleiner als 10 ms. Wie bereits angedeutet, ist EXECVM selbst ein Befehl, der bereits 0.02 bis 0.04 Sekunden braucht, das erklärt genau die Zeitdifferenz. Wir wollen damit darauf hinweisen, dass externe Skripte zwar die Lesbarkeit und Wartbarkeit erhöhen, gerade bei EH aber mit Bedacht eingesetzt werden sollten! Immerhin kann der Spieler ohne Probleme ein Magazin in wenigen Sekunden leerschießen und bei jedem einzelnen Schuss soll dann auch noch ein externes Skript geladen werden! Gut, dass wir bereits PREPROCESSFILELINENUMBERS hatten… 😉.

Übungen

[1] Eine Übung hatten wir euch bereits versprochen: Versucht, mit dem EH „Fired“ ein Skript zu starten, das feststellt, ob die Waffe/Munition schallgedämpft ist.

Hinweis: Da es hierfür keine direkte Lösung gibt, müsst ihr über den Klassennamen gehen. Dies ist bei Waffen leichter, da hier immer SD für schallgedämpft steht (bei Magazinen leider oft leider gemischt S oder SD, deswegen). Für das Suchen eine CBA-Funktion verwenden!

[2] Wir wollen dem Spieler ein futuristisches Navi schenken. Dazu soll ein EH gefunden werden, der detektiert, wann der Spieler fährt und wann er stehen bleibt. Das zugehörige Skript soll während der Fahrt die Geschwindigkeit, den Tankstatus sowie den Zustand regelmäßig mitteilen. Außerdem soll es über die verbleibende Entfernung zu einem feststehenden Missionsziel (z.B. einem Gebäude) informieren. Nachdem der Spieler das Fahrzeug abgestellt hat, soll das Navi über die Dauer der Fahrt unterrichten.

Hinweis: Eine anspruchsvolle Übung, also lieber mehrmals die Aufgabenstellung lesen. Der EH muss diesmal dem Fahrzeug zugewiesen werden. Eine Schleife stellt die Laufzeit während der Fahrt sicher, indem sie prüft, ob man fährt oder nicht. Die Fahrzeit kann über die Zeitdifferenz Fahrtende – Fahrtbeginn gebildet werden. Dafür braucht ihr den Befehl **time**. Für den Tank braucht ihr außerdem **fuel**.

I. 9. TIPPS & TRICKS SOWIE BESONDERHEITEN VON SQF

Gleich vorweg: Dieses Kapitel ist das vielleicht spannendste, wichtigste und gleichzeitig auch unwichtigste Kapitel von allen. Wir wollen damit sagen: Wer die hier vorgestellten Konzepte und Techniken beherzigt, holt auch das letzte Quäntchen aus ArmA und SQF heraus. Wer es nicht tut...wird auch überleben .[44]

Wir haben folglich in dieses Kapitel alles gestopft, was uns den Rahmen der anderen Kapitel zu sprengen schien. Ihr findet hier Vertiefungen, alternative Ansätze, Nice-to-Knows und sonstige Ergüsse in beliebigem Umfang. Wer will, kann einfach nur die Abschnitte lesen, die ihn interessieren (was ja hoffentlich immer gilt). Wir raten zum gezielten Herumstöbern und Ausprobieren. Dann fangen wir mal an.

Der professionelle Umgang mit STRINGs – format und parseText

Bisher wurde das Thema Textausgabe etwas stiefmütterlich behandelt und immer wieder aufgeschoben. Mit HINT und STR sind wir zwar an sich schon weit gekommen, es macht aber keinen wirklichen Spaß, permanent Strings und andere Datentypen mit „+“ zusammenzufügen.

Die Generallösung für all diese Probleme lautet **format**.

Description

Description:	Composes a string containing other variables or other variable types. Converts any variable type to a string. If you want to convert a string back to a number, use parseNumber

Syntax

Syntax:	String = **format** [formatString, var1, var2 ...]
Parameters:	[formatString, var1, var2 ...]: Array formatString: String - a string containing text and/or references to the variables listed below in the array. The references appear in the form of %1, %2 etc. var1: Any Value - variable referenced by %1 var2: Any Value - variable referenced by %2 .....And so on.
Return Value:	String - The full string is returned.

Zunächst handelt es sich bei FORMAT um einen Befehl, der verschiedene Ausdrücke mit unterschiedlichen Datentypen in einen STRING verwandeln kann. Der Rückgabewert ist ein einziger STRING und damit besonders für Bildschirmausgaben geeignet. Etwas gewöhnungsbedürftig ist allerdings die Syntax. Diese ist insofern einmalig in SQF als dass Referenzen auf nachfolgende Variablen in Form von Platzhaltern erlaubt sind. Das heißt konkret: Als erstes definiert ihr einen beliebigen STRING, der euren Ausgabetext beinhaltet (formatString). Die Werte allerdings, die ihr aus Variablen auslesen wollt, fügt ihr als Platzhalter mit dem Prozentzeichen und einem Index ein (%1, %2, usw.). Jetzt werden innerhalb des Arrays format[] nach dem Ausgabetext die referenzierten Variablen aufgelistet. Die erste hat den Index 1, die nächste den Index 2 usw. Damit könnt ihr also beliebig viele Variablen in einem einzigen STRING vereinen, ohne euch über den Datentyp Gedanken machen zu müssen. Da das ganze immer noch sehr abstrakt klingt, ein einfaches Beispiel:

```
hint format["Willkommen %1!", name player]
```

Diese einfache Begrüßung beinhaltet die wesentlichen Elemente von FORMAT. Zunächst legen wir unseren Text fest. Dabei ist die Variable, die wir auslesen wollen, der Spielername. Daher legen wir hier für den Platzhalter `%1` an. Die 1 ist wichtig, weil wir nach dem String nur eine Variable haben, daher muss hier zwingend eine 1 stehen, wenn wir darauf zugreifen wollen. Die Variable selbst ist eigentlich ein Code, denn der Befehl **name** ist eine Funktion, die den Anzeigenamen des Objektes zurückgibt, bei Spielereinheiten gerade den Spielernamen. Damit referenziert `%1` auf den Rückgabewert von NAME, das ist

[44] Demjenigen wird es aber schwerfallen, die Skriptarbeit von anderen professionellen Skriptern immer nachzuvollziehen.

ein STRING mit dem Spielernamen und dieser wird dann von FORMAT an der entsprechenden Stelle eingebaut:

Wir bemerken also bereits hier, wie unglaublich mächtig FORMAT ist. Nicht nur, dass es beliebige Datentypen verarbeiten kann, wir können die auszulesenden Werte auch direkt als Code angeben und brauchen keine lokalen Hilfsvariablen umständlich dazwischenzuschalten!

Eine ganz besondere Bedeutung bekommt FORMAT zusätzlich, wenn es darum geht, Zufallsvariablen oder fortlaufend nummerierte Variablen zu erzeugen. Nehmen wir einmal an, wir bräuchten ein dynamisches ARRAY, dessen Länge und dessen Einträge wir vorab nicht kennen. Z.B. weil wir jede Einheit mit einem Marker versehen wollen, die Anzahl der Einheiten aber nicht von Anfang an feststeht (typisches MP-Problem). In unserem Skript müssen wir an irgendeiner Stelle dieses ARRAY erzeugen und wir brauchen für jeden Spieler eine dynamische Variable, die z.B. den Namen des Markers darstellt. Wir wollen also ein ARRAY, das abhängig von der aktuellen Anzahl bestimmter Objekte oder Einheiten gefüllt wird. Dazu müssen wir Variablenbezeichner erzeugen, die dynamisch generiert werden, wir wissen ja vorher die Länge und damit auch die Zahlen nicht. Wir können zwar bisher mit COUNT zählen und ein ARRAY ohne Probleme mit SET oder „+“ um Elemente erweitern, aber wie erzeugen wir Variablennamen mit dynamischem Inhalt? Die erste Idee wäre natürlich eine Zählschleife wie die FOR-Schleife, die ja wunderbar eine Laufvariable regelmäßig hochzählt:

Listing_46.sqf
```
1  array = [];
2  for "_i" from 1 to (count playableUnits) do
3  {
4    array = array + ["var_marker"+str(_i)] // erzeugt ein dynamisches Array
5  }
```

Wir wollen – wie oben beschrieben – unser ARRAY `array` (hier global) also in Abhängigkeit der aktuellen Anzahl der spielbaren Einheiten mit STRINGs füllen, die alle unterschiedlich heißen sollen! Obige Idee würde dies an sich erfüllen, denn die Laufvariable `_i` zählt ja von 1 bis zur Anzahl der Spieleinheiten hoch, und der „+“ verbindet zwei Strings, wobei `str(_i)` ja genau eine Zahl als STRING liefert. Tatsächlich funktioniert die obige Lösung auch. Sie ist aber nicht sehr elegant. Mit dem neuen Befehl können wir daher auch schreiben:

Listing_47.sqf
```
1  array = array + [format["var_marker_%1",_i]];
```

Eigentlich sollte der erste Code gar nicht funktionieren, denn bisher hatte ich immer große Probleme, diese Lösung mit einer Verknüpfung durch „+“ zu realisieren, aber wie der Zufall es will, klappt es genau für dieses Handbuch natürlich doch 😵. Dennoch raten wir dringend zu Variante zwei, also der Erzeugung von dynamischen Bezeichnern mittels FORMAT, da die anderen Methoden stark fehleranfällig sind, auch wenn es hier funktioniert hat. Der Code ist kompakter, prägnanter, besser zu lesen, leichter zu warten und vor allem bei komplexeren Vorhaben auch benutzerfreundlicher. So könnte man ja auch mit zwei verschachtelten FOR-Schleifen arbeiten, um zwei unabhängige Laufvariablen zu generieren.

Ein ebenso mächtiges Einsatzgebiet des FORMAT-Befehls ist das dynamische Generieren von Code. Wir erinnern uns an die Befehle SPAWN und CALL, die beide in der Lage sind, einen Code, der als String vorliegt, auszuführen. Bisher hatten wir diesen Code aus einer externen Skriptdatei per PREPROCESSFILELINENUMBERS herausgelesen. Wir können aber auch eigenen oder vom Spieler eingegebenen Code ausführen lassen, indem wir FORMAT benutzen. Dazu müssen wir nur vorher natürlich den String, den FORMAT liefert, mit COMPILE kompilieren und schon kann es losgehen:

```
call (compile format["hint 'Willkommen %1'", name player]);
```

Diesmal haben wir also einen vollständigen Befehl innerhalb des format-Befehls. Diese Variante erscheint zunächst sicherlich ziemlich merkwürdig und wenig praxisrelevant. Tatsächlich braucht man natürlich solch eine Lösung nur dort, wo man ausführbaren Code als String vorliegen hat. Normalerweise schreiben wir unsere Skripte ja selber und brauchen dort auch keine solche Konstruktion. ArmA bietet aber auch die sehr umfangreiche Möglichkeit, Benutzerdialoge selbst zu schreiben und so die Interaktion zwischen Skript und Spieler auf ein höheres Niveau zu heben. Anstatt langer und umständlicher Aktionmenüeinträge oder Funkauslöser kann der Spieler dann z.B. Eingaben in ein Dialogfeld tätigen. Das können auch ganze Skriptbefehle sein. Diese werden dann von einem Skript ausgelesen und sollen natürlich als nächstes umgesetzt werden. Das Problem hierbei ist nur: Alles, was man aus Dialogen auslesen kann, ist fast immer vom Datentyp STRING. Wenn man sowieso einen Befehl benutzen möchte, der als Argument den Typ STRING erwartet (wie hint) ist das kein Problem. Wenn man aber z.B. die Eingabe als Code ausführen will, ist es das sehr wohl. Hat der Spieler z.B. die Eingabe `hint „Hallo an alle!"` getätigt, so müssen wir diesen STRING jetzt in Code umwandeln. Das ist allerdings mit dem COMPILE-Befehl kein Problem, da dieser ja einen STRING erwartet.

```
call compile "hint ""Hallo an alle!""";
```

Entweder speichern wir die Eingabe des Spielers in einer lokalen Variablen, oder wir übergeben den STRING direkt. Immer darauf achten, dass innerhalb eines STRINGs ein weiterer STRING in doppelten Anführungszeichen (""…"") oder in Apostrophen ('…') stehen muss!

Kniffliger wird es jetzt, wenn der Spieler nur einzelne Werte eingibt, die wir in einem Befehl brauchen. So ist es ja denkbar, dass der Spieler (wie bei Artillerie und Mörser) z.B. die Koordinaten für ein Ziel per Dialog eingibt. Dann stehen uns in unserem Skript die Koordinaten x, y und z als STRINGS zur Verfügung. Wenn wir jetzt diese Koordinaten in irgendeiner Form z.B. mit dem Befehl **createVehicle** verarbeiten wollen, hilft uns dabei FORMAT:

Listing_48.sqf

```
1  _x = ctrlText 101; // Auslesen einer Dialog-Textbox mit idc 101
2  _y = ctrlText 102; // Auslesen einer Dialog-Textbox mit idc 102
3  _z = ctrlText 103; // Auslesen einer Dialog-Textbox mit idc 103
4  call compile format["'HMMWV_M2' createVehicle [%1,%2,%3]",_x,_y,_z]
```

In den ersten drei Zeilen lesen wir die Eingabe vom Benutzer aus drei fiktiven Textfeldern aus. Damit haben wir drei separate STRINGs. Für die Erzeugung eines Objektes per CREATEVEHICLE brauchen wir aber eine Position vom Datentyp ARRAY mit x-, y- und z-Eintrag. Dazu verwenden wir jetzt FORMAT, denn damit können wir den Befehl als STRING schreiben und die geforderten drei Koordinaten einfach per Referenz übergeben. Somit haben wir interaktiv mit dem Spieler ein Fahrzeug gespawnt! Das Besondere hieran ist, dass dies geht, obwohl die Variablen `_x`,`_y` und `_z` vom Typ STRING sind! Normalerweisen hatten wir bisher nur Variablen, die vom Typ NUMBER waren. FORMAT verknüpft aber auch STRINGs miteinander und wandelt auch quasi den STRING „1" in die Zahl 1 um.

Damit haben wir den Befehl FORMAT weitgehend erschlagen und hoffentlich einige nützliche Anregung zu dessen Verwendung gegeben. Was uns jetzt noch fehlt, sind einige Worte zu einem anderen mächtigen Befehl: **parseText**.

PARSETEXT ist ein Befehl, der bestimmte XML-Attribute erlaubt und als Rückgabewert einen Structured Text erzeugt. Man kann sich dies so vorstellen, dass normale Texte, die nur in Anführungszeichen stehen, einfache, unstrukturierte Texte sind. PARSETEXT hingegen erzeugt Texte mit Struktur, insofern als dass nun auch Zeilenumbrüche, Bilder oder Schriftart und –farbe eingefügt werden können.

Tatsächlich bietet auch der einfache unstrukturierte Text einige Formatierungsmöglichkeiten wie z.B. „\n" als Zeilenumbruch. Hierbei handelt es sich aber um ein internes Steuerzeichen, das vom Compiler interpretiert wird und nicht um ein XML-Attribut.

Mit PARSETEXT werden auf einen Schlag wesentlich mehr Möglichkeiten freigeschalten:

Listing_49.sqf

```
1  _txt = parseText "First line<img image='data\isniper.paa'/><br/>Second line";
2  _txt = parseText "<t size='2' color='#ffff00'>Your yellow text!</t>";
```

Zunächst haben wir in der ersten Zeile eine Möglichkeit, mittels XML-Tags Bilder und Umbrüche einzufügen. Dabei sind sehr viele Elemente, die in XML oder HTML erlaubt sind, hier ebenfalls möglich. So können wir das IMG-Tag um das Attribut SIZE erweitern, um das eingebundene Bild noch in seiner Größe anzupassen (also `<img size='2' image='url' />`). Wichtig dabei sind folgende zwei generelle Merkregeln: Im Gegensatz zu HTML bzw. nach XHTML müssen Tags, die über kein schließendes Abschlusstag verfügen, mit einem Schrägstrich (/) beendet werden. Dies ist wichtig, da es hierbei sonst zu **keiner** Fehlermeldung kommt, der Text aber nicht weiter abgearbeitet wird! Außerdem werden die Attribut-Werte immer in Anführungszeichen gesetzt. Wir wissen aber bereits, dass Anführungszeichen innerhalb von STRINGs, also äußeren Anführungszeichen nicht einfach vorkommen dürfen. Daher müsst ihr entweder zu doppelten Anführungszeichen oder zu einfachen Hochkommata greifen. Im zweiten Beispiel habt ihr die Möglichkeit, eure Textaufgaben deutlich ansprechender zu gestalten. Damit ist es möglich, den Text in Schriftart, Größe, Ausrichtung, Farbe und Erscheinung genau anzupassen. Zunächst die beiden Beispiele als Screenshots:

Abschließend ein etwas umfangreicheres Beispiel, das euch die beiden neu gelernten Befehle FORMAT und PARSETEXT im Zusammenspiel präsentiert und wirklich tolle Ergebnisse für die Ausgabe zaubert. Dazu müssen wir noch zwei kurze Befehle besprechen.

Date ist ein Befehl, der als Rückgabeargument ein ARRAY mit 5 Elementen liefert. Die ersten 3 Einträge stehen für Jahr, Monat und Tag der Mission, wie sie in den Missionseinstellungen festgelegt worden sind. Eintrag 4 und 5 stehen dann für die Uhrzeit als Stunde und Minute. Jeder Eintrag lässt sich getrennt auslesen und damit gezielt neu zusammensetzen.

ComposeText ist ein notwendiger Befehl, der einfach den „+“-Operator für STRUCTUREDTEXT mimt. Mit dem Befehl PARSETEXT erzeugen wir nämlich nicht einen STRING, sondern den Datentyp **StructuredText**. Dieser beinhaltet ja auch einige Dinge, die ein normaler STRING nicht fassen kann (eben die XML-Befehle). Damit lässt sich aber auch nicht einfach der „+“-Operator auf zwei STRUCTUREDTEXTs anwenden. Stattdessen bedarf es des COMPOSETEXT-Befehls, der aber denkbar einfach funktioniert: Es werden einfach alle Einzeltexte (in Form von Hilfsvariablen) als ARRAY übergeben.

Listing_50.sqf

```
1   _nameEinheit = name player;
2   _anzahlEinheitenGruppe = count units group player;
3   _distanzZumZiel = floor (player distance ziel);
4
5   // date liefert:  array [year, month, day, hour, minute]
6   _year = date select 0;
7   _month = date select 1;
8   _day = date select 2;
9   _hour = date select 3;
10  _minute = date select 4;
11
12  _missionsBriefing = parseText format ["<t size='2' align='center'>
    Willkommen %1!</t>", _nameEinheit];
13
14  _trenner = parseText "<br/>**************************<br/>";
15
16  _missionsDatum = parseText format ["Es ist der %1.%2.%3 <br/> %4:%5 Uhr",
     _day, _month, _year, _hour, _minute];
17
18  _missionsZiel = parseText format ["Ihr Missionsziel ist <t
    color='#ff0000'>%1m</t> entfernt! <br/> Sie sind zu <t
    color='#00ff00'>%2.</t>", _distanzZumZiel, _anzahlEinheitenGruppe];
19
20  // Ausgabe aller Teilstrings
21  hint composeText[_missionsBriefing, _trenner, _missionsDatum, _trenner,
    _missionsZiel]
```

Leider sieht das Beispiel auf den ersten Blick recht abstoßend aus, das tut uns leid! Ihr müsst beim Lesen des Codes auf die Zeilenzugehörigkeit gut aufpassen, aufgrund der langen Texte sind die meisten Befehle nämlich über mehrere Zeilen gestreckt. Aber ihr erkennt ja das Ende einer Zeile stets am abschließenden Semikolon. Außerdem könnt ihr den Code in einen Editor eurer Wahl kopieren, was die Lesbarkeit dramatisch erhöhen sollte.

Wir definieren in den Zeilen 1-3 zunächst einige Hilfsvariablen, also den Namen des Spielers, seine aktuelle Gruppenstärke sowie die Distanz zu einem Ziel (hier exemplarisch ein Objekt mit dem Variablennamen `ziel`). Anschließend speichern wir alle Elemente des Befehls DATE in Hilfsvariablen, damit das spätere Zusammensetzen intuitiv von der Hand geht.

Ab Zeile 12 beginnen wir dann mit der eigentlichen Arbeit. Da wir logische Texthäppchen anbieten wollen, bietet es sich an, den Ausgabetext auch dementsprechend aufzuteilen. Daher definieren wir uns Hilfsvariablen, die jeweils nur einen Teil des Textes enthalten. Jede Einheit wird am Ende durch den `_trenner` voneinander abgetrennt, einfach einer Zeile mit Sternchen.

Innerhalb der einzelnen Ausgaben mischen wir nun bunt die Befehle PARSETEXT und FORMAT, um alle Informationen unterzubringen, die wir dem Spieler mitteilen wollen. Damit setzen wir den Text mal mittig, mal verändern wir Schriftfarbe und –größe. Selbstverständlich ist noch viel mehr möglich, was ihr aber alles im BI Wiki nachlesen könnt. Am Ende setzen wir alle Teiltexte mit COMPOSETEXT wieder zusammen. Das Ergebnis sieht dann wie folgt aus:

Performance und Dauer von Skriptbefehlen

Bis jetzt haben wir uns eigentlich ausschließlich immer nur um SQF-Befehle gekümmert. In diesem Kapitel wollen wir uns zur Abwechslung mit einem äußerst spannenden, interessant und daher für euch völlig akademischen Thema befassen: Der Performance von SQF-Befehlen. 95% der Leser können dieses Kapitel daher gefahrlos überspringen, es werden so gut wie keine neuen Befehle eingeführt. Für die restlichen 5 % könnte sich dieses Kapitel aber durchaus lohnen, denn wir werden versuchen, zumindest einige gängige Hinweise für eine bessere Performance von umfangreicheren Skripten zu geben.

Zunächst müssen wir natürlich definieren, was wir unter Performance verstehen. Mit Performance ist normalerweise die Leistungsfähigkeit und insbesondere in der Informatik die Datenverarbeitungsgeschwindigkeit gemeint. D.h. wir interessieren uns dafür, wie schnell bestimmte Befehle arbeiten bzw. wie schnell ein Skript vollständig abgearbeitet ist. Wir möchten hier ganz bewusst die Themen CPU-Belastung oder Speicherplatzbedarf ausklammern, da wir erstens nicht wüssten, wie man dies sinnvoll und effektiv messen könnte und es zweitens schlicht den Rahmen sprengen würde. Was wir tun können, ist uns über Geschwindigkeiten zu unterhalten, doch dafür brauchen wir eine Möglichkeit, diese zu messen.

Wir haben bereits den Befehl TIME kennengelernt. Dieser würde dafür ebenfalls völlig ausreichen. Da die Entwickler bei BI aber extra eine diag-Funktion eingebaut haben, wollen wir diese auch dankbar nutzen: **diag_tickTime**. Der Befehl ist denkbar einfach und gibt die Echtzeit wieder, die seit dem Spielstart vergangen ist, also dem Starten von ArmA und nicht dem Missionsstart! Das ist der Unterschied zu time. Außerdem sei der Vollständigkeit halber auch die magische Variable _TIME erwähnt, die nur innerhalb von Skripten existiert und die die Laufzeit des Skriptes misst.[45]

Die Zeitmessung ist nun denkbar einfach: Wir brauchen keinen Zähler kompliziert mitlaufen zu lassen, sondern wir nutzen simple Subtraktion: Wir speichern die Zeit zum Aufruf des Skriptes und an dessen Ende. Die Differenz aus diesen beiden Werten ergibt sodann die reine Laufzeit der SQF-Befehle, die wir auf ihre Performance hin testen wollen. Damit haben wir folgendes Grundgerüst:

Listing_51.sqf

```
1  _t0 = diag_tickTime;
2  … // Hier stehen alle zu testenden SQF-Befehle
3  _t1 = diag_tickTime;
4  player groupChat str(_t1 - _t0)
```

In Zeile 1 und 3 wird jeweils die Zeit in einer Hilfsvariable gespeichert. In Zeile 4 geben wir die Differenz dieser beiden Werte dann im Gruppenkanal bekannt. Alternativ kann hier jeder andere Kanal oder eine direkte Ausgabe mit HINT oder **hintSilent** realisiert werden.

Damit haben wir neben dem eigentlichen Skript bzw. SQF-Befehl sofort eine visuelle Rückmeldung über die Performance. Je kürzer dabei die ausgegebene Zeit, desto performanter – sprich schneller – arbeitet der Befehl. Schauen wir uns jetzt ein paar einfache Vergleichsbeispiele an.

Prinzipiell ist die Performance von einfachen Ausdrücken oder Anweisungen nebensächlich. Wie lange ein einzelner hint-Befehl dauert, ist von wirklich untergeordneter Bedeutung. Richtig interessant wird die ganze Sache erst dort, wo der Rechner eine Menge Arbeit leisten muss, z.B. bei sehr großen Schleifen. Hier haben wir einige Möglichkeiten im Kapitel I. 5.B kennengelernt. Schauen wir uns also einmal folgendes Beispiel an:

[45] Obgleich mir schleierhaft ist, wie man sie benutzen soll, da sie mit EXECVM und SPAWN nicht funktioniert.

Listing_52.sqf

```
1   // Variante 1
2   for "_i" from 1 to 100 step 1 do
3   { hint str(_i) }
4   // Variante 2
5   for [{_i = 0}, {_i <= 100}, {_i = _i + 1}] do
6   { hint str(_i) }
7   // Variante 3
8   _i = 0;
9   while {_i <= 100} do
10  { hint str(_i); _i = _i + 1 }
11  // Variante 4
12  _i = 0;
13  waitUntil {hint str(_i); _i = _i + 1; _i >= 100 }
```

Wir machen in allen vier Varianten sage und schreibe dasselbe, nämlich einmal von 0 bis 100 zu zählen! Das einzige, was wir dabei variieren, ist die Art der Schleife. Wir wählen entweder eine Zählschleife (FOR) in einfacher oder kompakter Syntax, eine Dauerschleife (WHILE) oder aber eine Warteschleife (WAITUNTIL). Soweit, so gut. Jetzt schauen wir uns die Ausführungsgeschwindigkeiten an. Dazu testen wir jede Variante nach der oben beschriebenen Weise.

alternative for-Schleife	kompakte for-Schleife	while-Schleife	waitUntil-Schleife
0-0.0009 s	0-0.0009 s	0.0009-0.0019 s	3.1 s

Was für Unterschiede! Die FOR-Schleife ist mit Abstand die beste – also performanteste – Schleife. Sie zählt mitunter so schnell, dass tatsächlich als Messung eine 0 herauskommt. Die WHILE-Schleife ist als Daumenregel ca. 2x so langsam wie die FOR-Schleife. Die WAITUNTIL-Schleife verhält sich demgegenüber etwas spezieller und braucht ganze 3 Sekunden, was daran liegt, dass ihr Prüfintervall deutlich länger ist. Jede Schleife hat nämlich sozusagen intern codiert ein Prüfintervall hinterlegt, das festlegt, wie oft die Schleife ihre Bedingung prüft. Die FOR-Schleife muss gar nicht prüfen, aber auch hier gibt es sozusagen ein Durchlaufintervall. Dieses ist offensichtlich extrem kurz festgelegt, so dass die FOR-Schleife fast augenblicklich 100 Ausführungen in unter einer Millisekunde realisieren kann. Die while-Schleife hingegen muss ihre Bedingung stets zunächst prüfen, ehe sie fortfahren kann. Dies tut sie scheinbar etwas langsamer als die FOR-Schleife, aber immer noch deutlich schneller als ein Mal pro Frame[46]. Die WAITUNTIL-Schleife ist nun ganz anders: Sie prüft lediglich einmal pro Frame ihre Bedingung! Das bedeutet, dass man ihr quasi beim Zählen zuschauen kann, denn ein Frame entspricht ca. 33 ms. Damit ergibt sich folgendes Fazit: Wenn ihr eine Schleife braucht, die schnell und effizient eine Aufgabe erledigt, nehmt, wann immer es geht, die FOR-Schleife. Wenn ihr aber eine Bedingung oft prüfen müsst und mehrere Skripte im Hintergrund laufen, dann nehmt, wann immer es geht, eine WAITUNTIL-Schleife. Übrigens ist die FOREACH-Schleife ähnlich performant wie die FOR-Schleife und gehört zu den besten Strukturen in ganz ArmA. So braucht eine FOREACH-Schleife mit 150 Einträgen nur ca. 0.001 ms. Die vordefinierten Arrays ALLUNITS usw. können also problemlos benutzt werden, um effizienten Code zu schreiben.

Das führt uns auch zu einem Performance-Hinweis der anderen Art: Schleifen müssen nicht immer so schnell oder oft laufen, wie SQF das vorsieht. Oft reicht eine Prüfung aller ein bis zwei Sekunden völlig aus. Dazu eignet sich der SLEEP-Befehl hervorragend. Einfach als letzte Anweisung innerhalb einer WHILE-Schleife ein `SLEEP 1` oder `0.5` und schon habt ihr statt einer Schleife, die aller Millisekunde prüft, nur noch eine, die pro Sekunde eine Prüfung der Bedingung vornimmt. Spart Rechenzeit und Speicherplatz!

[46] Mit Frame meinen wir hier ein Bild pro Sekunde. Habt ihr also 30 FPS (frames per second) im Spiel, so würde eine Schleife wie die WAITUNTIL-Schleife, die einmal pro Frame läuft, ca. alle 1/30 Sekunde laufen.

Andere Befehle sind, obwohl der Code sehr umständlich erscheint, so unglaublich schnell, dass sich alternative Strukturen gar nicht lohnen. So beträgt die Ausführungsgeschwindigkeit von zwölf IF-Abfragen über eine Variable genau 0 ms. Daher lohnt es sich nicht, eine umständliche SWITCH-CASE-Struktur aufzusetzen. Nur die Lesbarkeit profitiert hiervon (und das ist natürlich nichts Schlechtes). Selbst wenn innerhalb der IF-Bedingung Code zunähst ausgeführt werden muss, scheint dies weit schneller als in 1 ms zu geschehen. IF-Abfragen sind also extrem performant.

So ist zum Beispiel der veraltete CREATEVEHICLE-Befehl laut Internet rund 500 mal langsamer als der neuere **createVehicle Array**-Befehl.

```
_jeep = "HMMWV_M2" createVehicle (position player);
hint str(_jeep)
```

Obiger Code dauert ca. 0.04 ms. Der gleiche Befehl mittels Array dauert aber nach eigenen Testes ebenfalls mindestens 0.04 ms. Von der angeblichen 500-fachen Geschwindigkeit ist nichts zu spüren.

Daher wollen wir es in diesem Kapitel auch damit bewenden lassen. Das Wichtigste wurde anhand der Schleifen demonstriert und dies ist sicherlich auch der vorrangige Sinn von Performanceverbesserungen. Einzelne Befehle auf ihre Geschwindigkeit hin zu testen und zu vergleichen, erscheint weder sinnvoll noch relevant. Wer aber signifikante Ergebnisse vorzuweisen hat, kann sich gerne bei uns melden.

Selbst das Nachprogrammieren von Funktionen kann sinnvoll oder jedenfalls nicht wesentlich langsamer sein. Natürlich gilt die Faustregel: Eine vordefinierte Funktion in SQF funktioniert besser und arbeitet schneller als eine benutzergenerierte Lösung. Dennoch sind folgende zwei Beispiele gleichschnell:

Listing_53.sqf

```
1  // Ausgabe der Distanz über den Befehl distance
2  hint str(position player distance [0,0,0]);
3
4  //Ausgabe der Distanz über eine eigene Funktion/Berechnung
5  _x = position player select 0;
6  _dx = _x - 0;
7  _y = position player select 1;
8  _dy = _y - 0;
9  _distanz = sqrt(_dx^2+_dy^2);
10 hint str(_distanz)
```

Es kommt natürlich immer auf die Komplexität des Beispiels an, aber wir wollen uns hier ja nur einige Grundprinzipien vergegenwärtigen. Dass bei entsprechend großen Arrays und großen Zahlen die Werte deutlicher voneinander abweichen können, ist völlig klar. Hier erbringen beide Lösungen aber eine Ausführungsgeschwindigkeit von 0ms, also unterhalb der Messschwelle.

Daher wollen wir als Fazit festhalten, dass es bei gutem Code selten um die Ausreizung aller Millisekunden geht. Es zeichnet natürlich einen Experten von einem Amateurskripter aus, wenn er um die verschiedenen Arbeitszeiten und Geschwindigkeiten von SQF-Befehlen weiß, mehr aber auch nicht. Wirklich wichtig, um langfristig gute Skripte zu schreiben und mit anderen Skriptern zusammenarbeiten zu können, sind aber die Lesbarkeit und Wartbarkeit des Codes. Dazu gehört eine ordentliche Benennung der Variablen (selbstbezeichnend), ein sinnvoller, logischer Aufbau des Skriptes (Kommentare), die Vermeidung von Redundanzen (Schleifen) sowie Kompaktheit (Funktionen, kleine Skripte mit klaren Aufgaben).

Vorzeitiges Beenden von Schleifen und Skripten - Sprungstellen

Wir wollen uns jetzt einem Kapitel widmen, das zunächst einmal etwas exotisch erscheint, aber bei anspruchsvolleren Aufgaben sehr schnell an Bedeutung gewinnt. Die Rede ist von Abbruchbedingungen bzw. Abbruchbefehlen, um Skripte und Schleifen vorzeitig beenden zu können.

Prinzipiell sind uns Abbruchbedingungen ja schon z.B. in WHILE-Schleifen vertraut. Die Abbruchbedingung unterbindet eine Endlosschleife und sorgt dafür, dass die WHILE-Schleife abgebrochen wird, wenn die Bedingung quasi nicht mehr erfüllt ist. Im Falle der WHILE-Schleife formulieren wir sozusagen eine negative Abbruchbedingung, da wir ja im Kopf der Schleife festlegen, für welchen Fall sie laufen soll („solange, wie…"). Damit haben wir aber die Schleife noch nicht vorzeitig beendet und auch das Skript läuft danach natürlich noch weiter, sofern weitere Befehle nach dem WHILE-Block folgen. Wir interessieren uns also zunächst für zwei neue Fälle: Wie beende ich ein Skript, obwohl es noch nicht bis zur letzten Zeile abgearbeitet wurde (und warum möchte ich das überhaupt?) und wie beende ich eine Schleife, obwohl die Bedingung der Schleife noch erfüllt ist (und warum sollte ich das wollen?).

Der erste Fall – das vorzeitige Beenden eines Skriptes – haben wir tatsächlich schon kennengelernt: Der Befehl dafür lautet EXITWITH. Dieser Befehl ist eigentlich ursprünglich für den zweiten Fall gedacht gewesen, nämlich das Beenden von Schleifen, funktioniert aber außerhalb dieser genauso, nur beendet er dann eben das ganze Skript. Das Besondere an diesem Befehl ist seine geniale Eigenschaft, beim Beenden noch Code ausführen zu können, erkennbar an den geschweiften Klammern für die Argumente laut Syntax-Beschreibung. Wir haben diesen Befehl bereits im Beispielprojekt aus Kapitel [2] kennengelernt. Damit eröffnen sich uns sehr mächtige Möglichkeiten gerade im Multiplayer, da wir gezielt bereits zu Beginn eines Skriptes auswählen können, wer das Skript überhaupt ausführen darf. Wir stellen einige Standard-Konstruktionen vor, denen man oft in Skripten begegnet. Es sei aber gesagt, dass dies eventuell meist günstiger z.B. im Auslöser selbst definiert werden kann und sollte.

```
if (isServer) exitWith{};
if !(isServer) exitWith{};[47]
```

Diese beiden Standardzeilen aus vielen MP-Skripten dienen ganz zu Beginn des Skriptes dafür, das Skript sofort zu beenden, falls der Server (Fall 1) oder ein Spieler (Fall 2) das Skript aufruft. Es gibt viele Befehle, die für den Server weniger Sinn machen, z.B. Textausgaben mit HINT. Genauso gibt es aber auch Skripte, die nur auf dem Server ausgeführt werden sollen, weil sonst der Befehl statt einmal so oft wie Spieler auf dem Server sind ausgeführt wird. Klassisch hier sind spawn-Skripte oder Berechnungen bzw. Variablen, die nur einmal ermittelt und dann an alle Clients übermittelt werden sollen. Aber wie bereits erwähnt ist es oftmals auch für die Wartung besser, diese Dinge direkt in den Auslöser zu schreiben. Dort würde man dann für die beiden Fälle folgende Bedingungszeilen schreiben:

```
this && !isServer //Bedingungszeile für einen Auslöser nur für Spieler
this && isServer //Bedingungszeile für einen Auslöser nur für den Server
```

Der Vorteil: Auf dem Editor sind alle Auslöser zu sehen, allein mit dem Mauszeiger über einem Auslöser wird bereits ein Teil der Bedingung angezeigt. Damit kann ich sofort auf einen Blick alle relevanten Auslöser prüfen. Änderungen können notfalls sogar per Editor an der MISSION.SQM vorgenommen werden. Bei Skripten, die jeweils mit einer eigenen EXITWITH-Zeile beginnen, muss aber in einem Änderungsfall jedes Skript angepasst werden. Das wäre bei mehreren Dateien und verzweigter Ordnerstruktur dann doch recht aufwändig und eben unnötig.

Aber der EXITWITH-Befehl kann natürlich noch mehr als nur eine Fallunterscheidung für den MP vorzunehmen. Oftmals ist er wesentlich eleganter als z.B. eine IF-ELSE-Lösung. Möchten wir z.B. sicherstellen, dass der

[47] Man kann nicht oft genug darauf hinweisen, dass diese Befehle im SP nicht wie beschrieben funktionieren, da der Spieler selbst gleichzeitig auch der Server ist. Selbst wenn man ein Spiel im Internet selbst hostet, funktioniert ISSERVER nicht wie gedacht, da man selbst als Host der Server ist. Daher testet man solche Skripte am besten auf unabhängigen Servern.

Spieler bei Skriptausführung noch lebt und er Leader seiner Einheit ist, wäre prinzipiell folgender Code möglich:

Listing_54.sqf

```
1  if (alive player && leader player == player) then
2  {
3  ...
4  }
5  else
6  {
7  ... // der interessante Teil
8  };
```

Wir müssten folglich um unser gesamtes Skript eine IF-Bedingung setzen und ganz ans Ende scrollen, um zu sehen, was denn im Falle des Nichtzutreffens der Bedingung geschehen soll. Das ist leider nicht sehr lese- und wartungsfreundlich. Mit einem EXITWITH haben wir diese Probleme nicht:

Listing_55.sqf

```
1  if !(alive player && leader player == player) exitWith {...};
```

Vorteil: Der Hauptcode des Skriptes muss nicht mehr in einen THEN-Block gepackt werden, da wir diesmal eine IF-Schleife direkt mit dem Befehl EXITWITH verknüpfen. Dazu müssen wir natürlich die Bedingungen gerade negativieren, da wir ja jetzt die Bedingung für das Beenden des Skriptes angeben. Eine weitere Besonderheit ist, dass die Syntax für diese Zeile immer `if(bedingung) exitWith{anweisung};` lautet, also ohne THEN! Das führt oft zu dummen Fehlern. Außerdem sehen wir sofort, was im Falle des Beendens noch passieren soll, denn das drückt gerade die Anweisung im EXITWITH-Codeblock aus.

Das sollte für den Anfang genügen, um ein Gefühl für diesen Befehl zu erhalten und damit herumexperimentieren zu können. Wir wollen uns jetzt der Thematik des Beendens von Schleifen widmen.

Zunächst brauchen wir dafür gar keine neuen Befehle. Denn wie bereits erwähnt dient EXITWITH ursprünglich zum Beenden von Schleifen. Damit können wir also z.B. auch eine Schleife beenden, ohne uns vorher über die Abbruchbedingung direkt Gedanken zu machen. Zunächst ergibt sich daraus noch kein Vorteil, aber wir werden gleich sehen, warum dies sinnvoll sein kann.

Listing_56.sqf

```
1  while {true} do
2  {
3  if (time > 15) exitWith {…} //Abruch nach 15 Sekunden nach Missionsstart
4  };
5  … // weitere Anweisungen
```

In diesem einfachen Beispiel wollen wir zunächst die WHILE-Schleife ohne Abbruchbedingung starten, dazu wählen wir als Trick einfach die boolesche Variable TRUE. Damit ist die Bedingung immer erfüllt und die Schleife wird zu einer Endlosschleife. Damit habt ihr also auch gleich ein Beispiel, wie ihr ein Skript während der gesamten Mission im Hintergrund laufen lassen könnt. Diesmal erfolgt der Abbruch innerhalb der Schleife durch die `if (time > 15) exitWith{};`-Anweisung. Diese prüft jetzt statt der WHILE-Schleife, ob die vergangene Spielzeit größer als 15 Sekunden ist und beendet im Falle einer positiven Rückgabe die Schleife, und zwar nur die Schleife! Anweisungen, die nach den 15 Sekunden außerhalb der WHILE-Schleife folgen, werden in diesem Falle noch abgearbeitet, da EXITWITH nicht außerhalb einer Schleife steht. Da wir jetzt das Grundkonzept verstanden haben, können wir uns überlegen, warum so eine Konstruktion sinnvoll sein sollte. Denn im obigen Beispiel wäre es natürlich sinnvoller und für andere wesentlich einfacher nachzuvollziehen, wenn wir direkt

```
while {time < 15} do {…};
… // Anweisung nach dem Beenden, also was sonst in exitWith{} stünde!
```

geschrieben hätten. Sinn macht das ganze also nur dort, wo wir eine Abbruchbedingung brauchen, die wir nicht im Kopf der Schleife formulieren können! Mit anderen Worten: Meistens ergibt sich der Wert der zu prüfenden Variable erst innerhalb der Schleife:

Listing_57.sqf

```
1  while {driver auto == player} do
2  {
3  _distanz = auto distance ziel; //Distanz zum Ziel
4  _zufallsZahl = random 15;
5  if (_zufallsZahl + _distanz < 10 ) exitWith{hint "Sie sind quasi am Ziel"};
6  hint ("Ihr Ziel ist noch "+str(_zufallsZahl+_distanz)+ "m entfernt.");
7  sleep 2
8  }
```

Auch dies ist kein lupenreines Beispiel, weil wir auch hier natürlich die Abbruchbedingung im IF-Teil strenggenommen in den WHILE-Kopf schreiben könnten. Zwar kennen wir den konkreten Wert von `_zufallsZahl` nicht, aber wir hätten ähnlich gute Ergebnis bekommen, wenn wir im WHILE-Bedingungsteil noch ein `&& (auto distance ziel) < 10` hinzugefügt hätten. Der wahre Wert einer zusätzlichen Beendigungsbedingung zeigt sich erst bei komplexen Beispielen, wenn man z.B. Typensicherheit sicherstellen muss! So könnte es sein, dass während eines Benutzerdialoges im Hintergrund ständig ein Skript läuft, das die Eingabe des Nutzers auf Typenkonsistenz prüft und Alarm schlägt, sobald z.B. keine Zahl sondern versehentlich ein Buchstabe eingegeben wurde. In diesem Falle wäre ein EXITWITH erforderlich.

Neben diesem Beispiel ist es natürlich vor allem sinnvoll, EXITWITH dort einzusetzen, wo normalerweise gar keine Abbruchbedingung möglich ist, wie in einer FOREACH-Schleife.

Listing_58.sqf

```
1  {
2    if !(alive _x) exitWith {hint "Ein Missionsziel erfolgreich zerstört."};
3    hint str(getpos _x);
4    sleep 2
5  } forEach meinArray
```

In diesem Beispiel soll eine Schleife über alle Elemente des ARRAYS `meinArray` laufen, wobei dieses z.B. Missionsziele enthält (also Objekte mit entsprechenden Variablennamen). Wir möchten von jedem Missionsziel die Position auslesen und hier sogar dem Spieler ausgeben. Wenn aber ein Objekt gar nicht mehr zum Zeitpunkt der Abfrage aktiv/am Leben ist, dann können wir die Schleife vorzeitig verlassen. Auch kein besonders geglücktes Beispiel, wir entschuldigen uns für unsere mangelnde Fantasie in diesem Punkt.

Damit haben wir genug zu Schleifen gesagt. Was uns jetzt noch fehlt, ist die wohl mächtigste Variante, ein Skript vorzeitig zu beenden. Denn ein Problem gibt es mit EXITWITH immer noch: Es beendet lediglich den aktuellen **Scope**, zu dem es gehört. Das bedeutet vor allem, dass ich verschachtelte Schleifen so nur sehr schwer komplett beenden kann, da mehrere EXITWITHS notwendig wären. Hierfür gibt es eine elegantere Lösung: **breakOut**, **breakTo** und **ScopeName**. Diese drei Befehle sind eng miteinander gekoppelt und funktionieren nicht ohne eine sinnvolle Kombination von jeweils einem BREAK-Befehl mit einem SCOPENAME-Befehl. Das besondere an diesen Befehlen ist, dass damit Sprungverweise kreiert werden können (ähnlich wie früher in SQS), die dann mit BREAKTO direkt angesteuert werden können, bzw. mit BREAKOUT direkt verlassen werden können. Wir wollen an dieser Stelle das offizielle Beispiel aus dem BI Wiki besprechen:

Listing_59.sqf

```
1  scopeName "main"; // äußerer Scope
2  while {true} do
3  {
4    scopeName "loop1";
5    while {true} do
6    {
7      scopeName "loop2";
8      if (condition1) then {breakTo "main"}; // Breaks all scopes, return to
       "main"
9      if (condition2) then {breakOut "loop2"}; // Breaks scope named "loop2"
10     sleep 1
11   };
12   sleep 1
13 }
```

Zunächst werden in den Zeilen 1, 4 und 7 drei sogenannte Scope-Bereiche festgelegt. Diese dienen als Sprungmarken bzw. Referenzmarken für die Befehle BREAKTO und BREAKOUT. Man kann sich also den Befehl SCOPENAME wie zwei geschweifte Klammern vorstellen, die einen Block einleiten und dieser Block hat dann eben einen selbstgewählten Namen, mit dem man ihn ansprechen kann. Dabei geht ein Scope immer von einem SCOPENAME- bis zum nächsten SCOPENAME-Befehl. Möchte man nun den aktuellen Scope-Bereich vorzeitig verlassen, also quasi ein EXITWITH durchführen, das aber für den gesamten benannten Bereich gilt und nicht nur für eine innere Schleife, so wendet man den Befehle BREAKOUT an. Damit wird der aktuell gültige Scope-Bereich verlassen und quasi beim nächsten SCOPENAME weitergemacht. SCOPENAMEs können aber auch verschachtelt werden, obwohl das BI Wiki dies eigentlich verneint. Aber das Beispiel zeigt ja selbst, dass jeder Scope quasi im Inneren des vorherigen Scopes verschachtelt ist. Leider ist nie ganz klar, bis wohin ein SCOPENAME geht, da keine Klammern benutzt werden. Daher ist eine sehr saubere Programmierung und Kommentierung notwendig, wenn diese Befehle eingesetzt werden sollen.

Das Beispiel funktioniert nun so, dass die äußerste WHILE-Schleife eine Endlosschleife ist. In ihrem Inneren wird eine zweite WHILE-Schleife, ebenfalls ohne Abbruchbedingung, initiiert. Innerhalb dieser folgt quasi als dritte Stufe der Verschachtelung eine doppelte IF-Abfrage. Im Falle der ersten Bedingung (`condition1`) wird der Befehl `breakTo „main“` ausgeführt, was gleichbedeutend mit einem Abbruch aller aktuellen Schleifen und einem Sprung zur Marke `„main“` ist, also der 1. Zeile des Skriptes. Mit anderen Worten: Dieser Befehl führt dazu, dass im Programm zurückgesprungen und das komplette Skript ab Zeile 1 erneut ausgeführt wird. Die zweite Bedingung (`condition2`) führt im Falle einer positiven Auswertung zum Abbruch der innersten Verzweigung, also zu einem Beenden des Scopes `„loop2“`. In diesem Falle würde `„loop1“` aber noch aktiv sein und das Skript würde aufgrund der Endlosschleife erneut mit `„loop2“` beginnen.

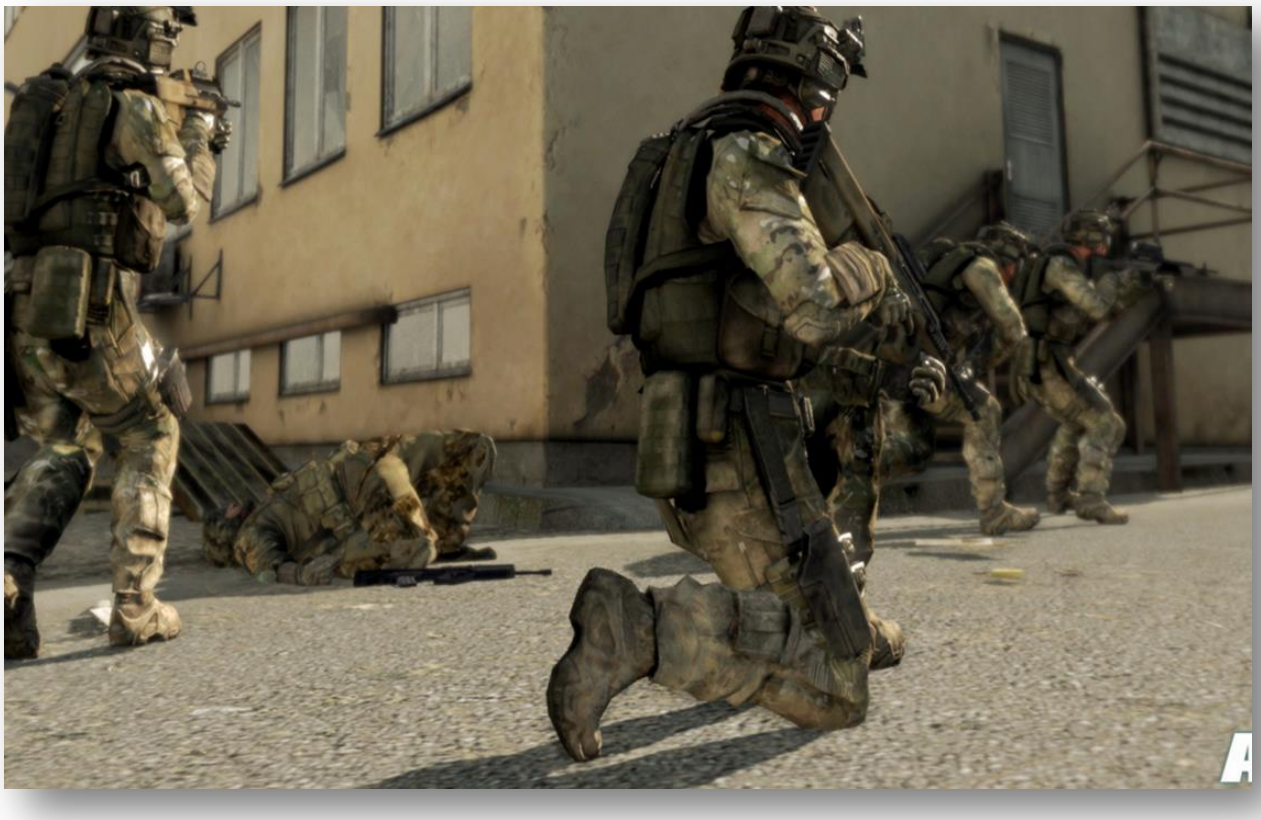
Wichtig an diesem zugegebenermaßen sehr abstrakten Beispiel ist eigentlich nur die mächtige Fähigkeit, mit SCOPENAME Bereiche benennen zu können, die dann mittels BREAKOUT komplett übersprungen werden können. Dies ist bei Fehlerbehandlungen wie z.B. falschen Zahlenformaten oder falschen Benutzereingaben sehr wichtig und kann mit diesen jetzt neu gelernten Befehlen beliebig komplex gestaltet werden.

Event Scripts

In diesem Kapitel wollen wir uns mit einigen wichtigen Skripten beschäftigen, den sogenannten Event-Scripts. Davon gibt es einige, aber was bitteschön sollen das für Skripte sein? Nun, erinnern wir uns doch noch einmal kurz an das Kapitel I. 8 „Event-Handler“ (man wird sich hoffentlich noch erinnern können…). Dort hatten wir eine neue Klasse von Befehlen kennengelernt, die immer dann „feuerten“ oder aktiviert wurden, wenn bestimmte Ereignisse in Kraft getreten waren. Genau das gleiche Prinzip gibt es nun in ArmA2. Diesmal brauchen wir aber keine Befehle, sondern das Spiel selbst achtet auf gewisse Ereignisse (Events), bei deren Eintreten es automatisch bestimmte Skripte startet. Und um diese geht es in diesem Kapitel.

SQS und SQF – Konvertierung

Teil II – SQF im Multiplayer



EINLEITUNG

II. 1. KLASSENNAMEN HERAUSFINDEN

II. 2. LOKALITÄT – DAS PROBLEM MIT DER GÜLTIGKEIT

Teil III – SQF in der Praxis

EINLEITUNG

Teil III – welch langer Weg liegt hinter uns! Vorausgesetzt natürlich, ihr seid nicht direkt hierher gesprungen 😵. In diesem Kapitel wollen wir uns einige Skripte, die in der Praxis Verwendung finden und ihre Tauglichkeit bewiesen haben, vorstellen. Natürlich müssen wir an dieser Stelle die Erwartungen allerdings etwas dämpfen: Auch wenn die obigen Bilder zu Teil III so etwas andeuten, werden wir doch keine Modifikationen oder Addons selber schreiben können. Für ACE reichen nicht einmal unsere eigenen Fähigkeiten, da lassen wir uns dann lieber von den richtigen Profis weiter belehren.

Was wir in diesem Kapitel aber anbieten können, ist die Durchsicht, die Besprechung und kritische Beleuchtung von einigen umfangreicheren Code-Beispielen. Sinn & Zweck dieses Kapitels soll es sein, all das bisher Gelernte zu festigen, weiter zu verfeinern und in bisher vielleicht nicht bekanntem Maße oder Kontext einzusetzen. Welche Lösungen wurden für welche Probleme gefunden? Welche bisherigen Konzepte sind dabei nützlich? Welche manchmal sogar hinderlich? Wie kann ich selbst ein Projekt realisieren und Fehler vermeiden?

Dieses Kapitel ist also eher eine Art Workshop mit gutgelauntem Kommentator. Wir werden uns Beispiele ansehen, sie auseinandernehmen, ihre Funktionsweise durchleuchten und uns am Ende fragen, ob sie das, wofür sie geschrieben wurden, auch tatsächlich leisten. An einigen Stellen werden wir auf noch bestehende Probleme stoßen, die ihr dann z.B. in Eigenregie und nach Lust und Laune selbst versuchen könnt zu beheben. In diesem Sinne hoffen wir, dass wir dem Leser damit entgegenkommen. Für den Profi wird es an der ein oder anderen Stelle langatmig werden, eventuell werden Beispiele auch nicht sofort als Endlösung präsentiert, sondern erst mit dem Leser entwickelt. Wir bitten dann um Verständnis für diejenigen, denen diese Vorgehensweise hilft, und raten allen anderen, die sich langweilen, direkt ans Ende der Kapitel zu den fertigen SQF-Dateien zu springen, denn wie immer wurden die Code-Beispiele als Screenshoots eingefügt und mit den originalen SQF-Dateien verlinkt, so dass alle Beispiele als Download zur Verfügung stehen.

Da dieses Kapitel ein loses Sammelsurium verschiedenster Skripte ist, gibt es keine Reihenfolge und erst recht keinen Anspruch auf Vollständigkeit. Wir ermuntern daher alle Leser, uns eine Nachricht zukommen zu lassen, wenn Sie Skripte vermissen oder selbst Vorschläge für dieses Kapitel hätten. Eine PDF-Datei ist zum Glück kein Buch und daher beliebig erweiterbar, und je mehr Beispiele hier zusammenkommen, desto besser können zukünftige Skripter von ihnen lernen. In diesem Sinne wünschen wir allen viel Spaß und viele Aha-Momente beim Lesen.

TABELLE 1: ÜBERSICHT ÜBER ALLE BESPROCHENEN BEISPIELE UND DEREN SCHWERPUNKTE

Kapitel/Beispiel	Behandelte Inhalte
III. 1 Klassennamen herausfinden	
III. 2 Nachschub mit Einschränkungen	Event Scripts, Event Handler, MP-Tauglichkeit, Vorzeitiges Beenden von Schleifen, switch case
III. 3 Bordfunk in Fahrzeugen	Event Handler, MP-Tauglichkeit, dynamische Arrays
III. 4 Gegner anhalten oder zerstören/unschädlich machen	

III. 1. KLASSENNAMEN HERAUSFINDEN

III. 2. NACHSCHUB MIT EINSCHRÄNKUNGEN

Respawnskripte gibt es sehr viele, ACE bietet sogar ein Modul, das den Respawn weitestgehend vereinfacht. In der `description.ext` finden wir alle wichtigen Einstellungen, um den Respawn für Einheiten und Fahrzeuge einzustellen und einen einfachen Respawn im MP zu ermöglichen. Auf der folgenden Seite sind die wichtigsten Infos gut zusammengefasst (leider auf Englisch, das sollte aber kein Problem sein):

http://sandbox.darrenbrant.com/arma_ii/how-to-setup-respawn-in-an-arma2-mission

In diesem Kapitel wollen wir aber unser eigenes Respawn-Skript schreiben, das folgende Auflagen erfüllt:

- Der Respawn soll auf drei festgelegten Nachschubpositionen erfolgen, z.B. *ingame* markiert durch weiße Helipad-Markierungen
- Der Respawn soll dabei limitiert sein. Da drei Positionen zur Verfügung stehen, können auch drei Anfragen auf Nachschub gestellt werden. Eine vierte Anfrage muss aber abgelehnt werden.
- Außerdem soll ein Respawn nicht mehr möglich sein, wenn zwar drei Anfragen fertig bearbeitet wurden, die Fahrzeuge aber noch auf den Positionen stehen, also noch nicht abgeholt wurden.
- Der Respawn soll nur von bestimmten spielbaren Einheiten, namentlich dem Kommandanten und dem Nachschuboffizier, genutzt werden dürfen.
- Die Fahrzeuge sollen ihren Variablennamen auch nach dem Respawn behalten sowie mit einer fertigen Init-Zeile starten, so dass z.B. Loadouts auch nach dem Respawn wieder verfügbar sind.

Ich würde meinen, das ist eine beeindruckende Liste, die uns einiges an Arbeit abverlangen wird. Diesmal ist der Code außerdem auf drei Dateien verteilt: die `init.sqf`, die `vehicle_respawn_init.sqf` sowie das eigentliche Skript `vehicle_respawn.sqf`. Ein Ablaufdiagramm soll die Vorgänge verdeutlichen. Wir empfehlen generell zuerst immer eine Skizze oder einen Ablaufplan bei komplexeren Programmieraufgaben, ehe mit der eigentlichen Arbeit begonnen wird. Auf diese Weise spart man sich das permanente Umschreiben des Codes, weil man Kleinigkeiten übersehen oder schlichtweg vergessen hat. Es muss nicht gleich ein UML-Diagramm sein, ein einfaches Ablauf- oder Anwendungsfalldiagramm genügt in den meisten Fällen, um die Übersicht zu behalten. Gehen wir die Schritte der Reihe nach einzeln durch:

[1] Direkt bei Missionsstart spielt die `Init.sqf` eine wichtige Rolle. Sie bereitet nämlich alles weitere vor, indem sie das Skript `vehicle_respawn_init.sqf` aufruft. Dabei haben wir bereits die Möglichkeit, bestimmte Spieler auszuschließen. Hier wäre auch das Speichern des Skriptes mit `compile preprocessFile` möglich, wir werden aber später sehen, warum das nicht geht.
Außerdem wird in der `Init.sqf` für jeden Spieler ein PublicVariableEventHandler angelegt.

[2] Das aufgerufene Skript vehicle_respawn_init.sqf sorgt jetzt für alle wichtigen Einstellungen. Es werden die Aktionmenüeinträge an die beiden Fahrzeuge übergeben, die später sozusagen als Nachschubanforderungszentralen fungieren sollen. Außerdem wird das ARRAY `resapwn_array` angelegt, das die möglichen Positionen für den Respawn enthält. Des Weiteren finden sich hier die Init-Zeilen für die Fahrzeuge, damit auch nach dem Respawn Loadouts zur Verfügung stehen. Damit ist dieses Skript der zentrale Anlaufpunkt für das eigentliche Respawn-Skript und hier werden auch alle Einstellungen vorgenommen. Das eigentliche Skript ist damit unabhängig und muss nicht angepasst werden (außer es sollen weitere Fahrzeugtypen hinzugefügt werden).

[3] Damit sind die Vorbereitungen fertig und das Skript kann seine Arbeit aufnehmen. Ausgelöst wird es durch zwei bestimmte Spieler, die von uns die Erlaubnis bekommen, auf die Aktionmenüeinträge zugreifen zu können. Sobald sie dies tun, haben sie die Möglichkeit, aus voreingestellten Möglichkeiten eine Nachschuboption auszuwählen. Sobald sie einen Eintrag im Aktionsmenü mit 

bestätigen, wird das Hauptskript mit den entsprechenden Einstellungen aufgerufen, die eingestellt worden sind (also Übergabeparameter). Außerdem sind die Aktionsmenüeinträge so eingestellt, dass sie nur für die zwei berechtigten Spieler sichtbar sind.

[4] Wie bereits beschrieben ruft der ausgewählte Aktionsmenüeintrag das Hauptskript auf und übergibt je nach Auswahl den Fahrzeugtyp, der respawnt werden soll.

[5] Das Hauptskript hat zunächst die Aufgabe, aus den drei zur Verfügung stehenden Nachschubplätzen einen auszuwählen, auf dem die Anfrage durch den Spieler dann ausgeführt werden soll. Diese Aufgabe kann zwei Ausgänge haben: Entweder wird ein freier Platz gefunden, dann kann das Fahrzeug je nach Übergabeparameter auch dort respawnt werden. Oder aber alle drei Plätze sind belegt bzw. durch den anderen Spieler ist der letzte freie Platz bereits mit einer Anfrage belegt, so dass momentan keiner der drei Plätze frei ist. Dann wird das Skript beendet und der Spieler über die fehlenden Kapazitäten informiert. So oder so muss das Skript aber irgendwie die zur Verfügung stehenden Plätze managen, korrekt reduzieren und am Ende wieder freigeben, damit immer alle drei Plätze nach erfolgter Arbeit zur Verfügung stehen.

Das also wird unsere Aufgabe sein.

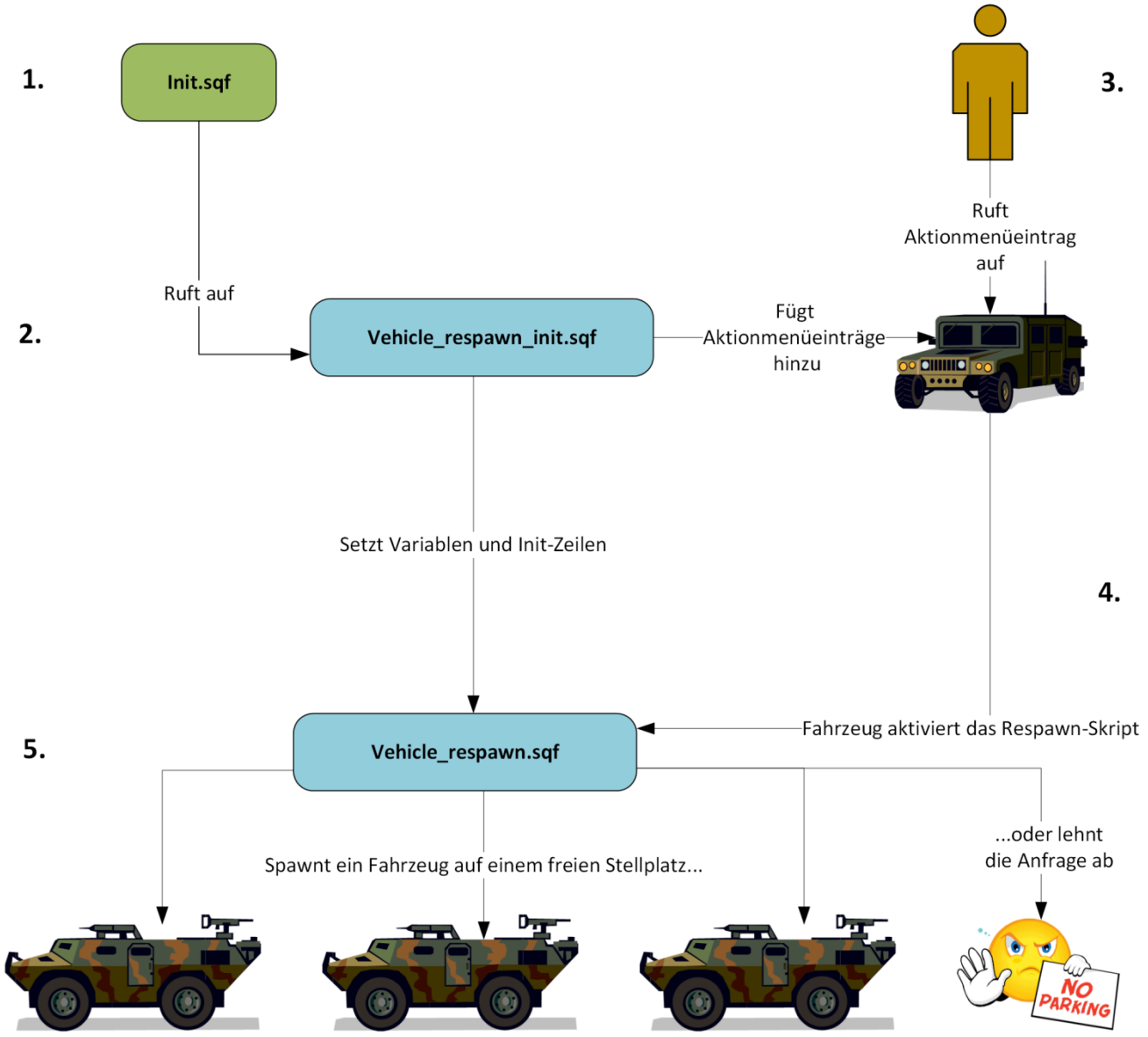


ABBILDUNG 1: ABLAUFDIAGRAMM

Damit haben wir alle konzeptuellen Überlegungen abgeschlossen. Wir haben eine genaue Auflistung der Aufgaben, die unser Skript erfüllen soll. Wir haben uns über den Ablauf Gedanken gemacht und das Skript bereits auf drei Teilskripte aufgeteilt. Damit können wir im Grunde loslegen und die einzelnen Bausteine zusammensetzen. Im Folgenden werden wir weitgehend anhand obiger Ablaufskizze vorgehen und das Skript Stück für Stück zusammensetzen.

Ganz wichtig: Für das folgende Beispiel müsst ihr einige Dateien umbenennen, wenn die Skripte auch bei euch laufen sollen. Dies ist notwendig, da ich nicht immer alle Dateien als Init.sqf abspeichern kann. Daher haben die Dateien Präfixe, die ihr natürlich entfernen müsst, damit die Skripte so funktionieren:

- Die Example_Vehicle_init.sqf → in Init.sqf,
- Die Example_Vehicle_vehicle_respawn_init.sqf → in vehicle_respawn_init.sqf und
- Die Example_Vehicle_vehicle_respawn.sqf → in vehicle_respawn.sqf umbenennen.

Wie ihr merkt, ist die Regel einfach: Einfach den Namen löschen, den ich allen Skripten vorangestellt habe, in diesem Fall also "Example_Vehicle". Zudem müssen die Dateien natürlich entsprechend in den Ordner `scripts` verschoben werden, wenn ihr meine Beispiele 1:1 übernehmt.

Die Init.sqf

Wir beginnen mit der wohl wichtigsten Datei in MP-Partien, der `Init.sqf`. Wie bereits in Teil I und Teil II beschrieben, übernimmt die `Init.sqf` als Event Script die Aufgabe, alle möglichen Einstellungen, Variablen und Skriptaufrufe noch vor dem eigentlichen Missionsbeginn zu laden/zu initialisieren. Die `Init.sqf` wird für gewöhnlich dazu verwendet, das Briefing zu starten und die Ausrüstung der Spieler festzulegen, ACE-Einstellungen zu treffen und missionseigene Variablen wie Aufträge oder Wahrheitswerte festzulegen. Wir zeigen nur den Ausschnitt, der für uns relevant ist und überlassen es euch, die `Init.sqf` mit für eure Mission wichtigen Befehlen weiter zu füllen:

Example_Vehicle_init.sqf

```
1  if (!isServer && player != player) then
2  {
3  waitUntil {player == player}
4  };
5
6  // Aufruf der wichtigen Skripte zu Missionsbeginn
7  // Notwendig, damit das Interaktionsmenü der Kiste korrekt erscheint
8  sleep 2;
9  _james_vehicle_init = [] execVM "scripts\vehicle_respawn_init.sqf";
10 waitUntil {scriptDone _james_vehicle_init};
11
12 //Ermöglicht bei jedem Spieler das Einstellen der ViewDistance bis Limit
   (5000) und des Grases
13 ace_settings_enable_vd_change = true;
14 ace_settings_enable_tg_change = true;
15
16 ////////////////////////////// Respawn-Script //////////////////////////////
17
18 // Das eigentliche Script wird bereits hier vorgeladen
19 //james_respawn_script = compile preprocessFile "scripts\respawn.sqf";
20
21 ////////////////////////////////////////////////////////////////////////////
22
23 // Fügt jedem Spieler einen Public-Event-Handler hinzu, der dann aktiviert
   wird, wenn die Variable eine Änderung erfährt
24 "global_exe" addPublicVariableEventHandler {call compile (global_exe)};
25
26 if(true) exitWith {hint "Gefechtsbereit"}
```

Nun ja, ein paar „unnötige" Zeilen sind doch mit reingerutscht, aber sie verdeutlichen damit nur den normalen Inhalt einer `Init.sqf`. In den Zeilen 1 bis 4 sehen wir den typischen Kopf für ein MP-Skript, bei dem wir möchten, dass alle Spieler korrekt erkannt werden und im Spiel sind, ehe das Skript weiter

arbeitet. In Zeile 9 erfolgt dann der eigentliche Aufruf der `vehicle_respawn_init.sqf`. Der Rückgabewert bzw. der Skript-Handler wird in einer Variable gespeichert, um den Status via SCRIPTDONE abfragen zu können. Wir könnten auch an dieser Stelle bereits mit einer IF-Abfrage dafür sorgen, dass nur die zwei berechtigten Spieler die Zeile 9 ausführen und damit für alle anderen diese Zeile einfach überspringen, aber wir denken, es ist für die Synchronisation besser, wenn alle die gleiche Zeit für den Skriptstart aufwenden müssen. Daher erfolgt die IF-Abfrage bezüglich der zwei erlaubten Spieler im Skript `vehicle_respawn_init.sqf` und nicht hier.

Die Zeilen 13 und 14 sind wieder Standardzeilen und erlauben einige Einstellungen im ACE-Menü.

Interessanter ist die Zeile 19, die auskommentiert ist. Prinzipiell wissen wir ja bereits, dass es bei häufiger verwendeten Skripten sinnvoll und geboten ist, diese zu kompilieren und in einer Variablen zu speichern. Wird das Skript dann irgendwann einmal gebraucht, kann es mit `call skriptVariable` einfach aufgerufen werden. Der Grund, warum wir in diesem speziellen Fall nicht mit dieser tollen Lösung arbeiten können, liegt darin, dass wir den Befehl **addAction** verwenden wollen, um Aktionsmenüeinträge zu erzeugen, die der Spieler dann nutzen können soll. Wie wir gleich sehen werden, erwartet dieser Befehl aber zwingend als zweites Argument eine Pfadangabe zu einem Skript. Daher würden wir mit kompiliertem Code nicht sehr weit kommen, wir müssen tatsächlich in den sauren Apfel beißen.

Fehlt uns noch die Zeile 24, die ganz entscheidend für viele komplexere Aufgaben im MP ist. Wir haben dieses Thema bereits ausführlich in Teil II behandelt, frischen aber hier das Gesagte noch einmal auf. Es gibt viele Befehle, die eine globale Wirkung erzielen sollen. Leider sind aber ebenso viele Befehle in ArmA lokal. Das bedeutet, ihre Wirkung gilt nur auf dem Client, auf dem sie aufgerufen/ausgeführt werden. Ein Beispiel hierfür ist der Befehl ADDACTION. Dieser erzeugt einen Aktionsmenüeintrag an einem Objekt. Dieser Eintrag ist aber nur für den Spieler sichtbar, auf dessen Rechner der Befehl ausgeführt wurde. Das fällt bei der Verwendung in der Init.sqf nicht auf, da dieses Skript ja von allen Spielern ausgeführt wird. Lassen wir aber den Befehl ADDACTION über einen Auslöser oder ein lokales Skript auslösen, so haben wir das eben genannte Problem. Es gibt verschiedene Möglichkeiten, dafür zu sorgen, dass ein Befehl definitiv auf allen Clients ausgeführt wird. Eine Möglichkeit ist die Verwendung der Funktion CBA_FNC_GLOBAL. Eine andere Möglichkeit ist die Verwendung von PublicVariableEventHandlern. Diese funktionieren wie normale EH, haben aber die Besonderheit, dass sie nur auf ein einziges Ereignis reagieren, nämlich auf den Befehl **publicVariable**. Wie wir ja bereits wissen, wird mit PUBLICVARIABLE der Wert einer Variablen an alle Clients und den Server übermittelt. Genau in diesem Moment löst der PublicVariable-EH aus, sofern er auf dem Client eingerichtet wurde. Die besondere Stärke liegt nun darin, dass wir jeden beliebigen Code mit der übermittelten Variable ausführen können, also z.B. auch den Befehl CALL COMPILE. Handelt es sich bei der übermittelten Variable um einen korrekten Befehl laut SQF-Syntax können wir damit also Code-Befehle an alle Clients übermitteln! Damit das funktioniert, muss aber jeder Spieler über einen PublicVariable-EH verfügen und das stellen wir in der `Init.sqf` sicher. Unsere PUBLICVARIABLE heißt `global_exe`. Dieser werden wir im Hauptskript `vehicle_respawn.sqf` dann auch wiederbegegnen.

Die Vehicle_respawn_init.sqf

Damit haben wir genügend Worte über den ersten Schritt verloren und gehen über zum nächsten, dem zweiten Schritt: der `vehicle_respawn_init.sqf`. Aufgrund der Länge musste ich allerdings zwei Screenshots machen.

Example_Vehicle_vehicle_respawn_init.sqf

```
1  // Bereitet den eigentlichen Respawn vor - by James
2
3  // Beendet das Skript an dieser Stelle für alle, die nicht die genannten
   Spieler sind
4  if (player != s1h && player != s1z) exitWith{};
5
6  // Array für alle Respawnorte! Bitte hier die Namen eure Respawnmöglichkeiten
   eintragen (unsichtbare Hs)
7  respawn_array = [respawn_vehicle_1, respawn_vehicle_2, respawn_vehicle_3];
8
9  // fügt den beiden Führungs-Fahrzeugen (WOLF) jeweils Action-Menüeinträge
   hinzu.
10 //Number = unitName addAction [title, filename, (arguments, priority,
   showWindow, hideOnUse, shortcut, condition, positionInModel, radius,
   radiusView, showIn3D, available, textDefault, textToolTip)]
11 {
12 ID_spz = _x addAction ["<t color='#ffffff'>SPZ anfordern</t>",
   "scripts\respawn.sqf",["SPZ"],0,true,true,"","_this == s1h || _this == s1z" ];
13 ID_lkw_mun = _x addAction ["<t color='#ffffff'>5t Munition anfordern</t>",
   "scripts\respawn.sqf",["LKW_MUN"],0,true,true,"","_this == s1h || _this ==
   s1z" ];
14 ID_lkw_rep = _x addAction ["<t color='#ffffff'>5t Reparatur anfordern</t>",
   "scripts\respawn.sqf",["LKW_REP"],0,true,true,"","_this == s1h || _this ==
   s1z" ];
15 ID_lkw_fuel = _x addAction ["<t color='#ffffff'>5t Treibstoff anfordern</t>",
   "scripts\respawn.sqf",["LKW_FUEL"],0,true,true,"","_this == s1h || _this ==
   s1z" ];
16 ID_10t = _x addAction ["<t color='#ffffff'>Munition anfordern</t>",
   "scripts\respawn.sqf",["10t"],0,true,true,"","_this == s1h || _this == s1z" ]
17 } forEach [wolf1, wolf2];
18
```

```
19  // Init der Fahrzeuge, brauchen nur die, die auch Zugriff auf Respawn haben,
    da alle Scripte per Actionmenü lokal ablaufen!
20  init_spz = "this AddMagazineCargo [""ACE_DM16_M"", 5];  this addMagazineCargo
    [""ACE_20Rnd_762x51_B_G3"", 40];  this addMagazineCargo [""SmokeShell"",
    10];  this addWeaponCargo [""ACE_Backpack_Olive"", 9];  this addMagazineCargo
    [""ACE_DM51A1"", 24];  this addWeaponCargo [""BWMod_PzF"", 1];  this
    addMagazineCargo [""BWMod_PzF_3"", 5];  this addMagazineCargo
    [""1Rnd_HE_M203"", 20];  this addMagazineCargo [""BWMod_MG3Mag"", 10];  this
    addMagazineCargo [""ACE_LargeBandage"", 10];  this addWeaponCargo
    [""ACE_M220Proxy"", 1];  this addWeaponCargo [""ACE_M220TripodProxy"", 1];
    this addMagazineCargo [""ACE_TOW_CSWDM"", 4]; this addEventHandler
    [""GetIn"", {[_this select 0, _this select 2] execVM
    ""scripts\bordfunk.sqf"";}];";
21  init_lkw = "ClearMagazineCargo this;  this addMagazineCargo
    [""ACE_20Rnd_762x51_B_G3"", 10];  this addMagazineCargo [""SmokeShell"", 2];
    this addMagazineCargo [""ACE_LargeBandage"", 4];this addWeaponCargo
    [""ACRE_PRC117F"", 1];";
22  init_10t = "ClearWeaponCargo this; ClearMagazineCargo this; this
    addMagazineCargo [""ACE_20Rnd_762x51_B_G3"", 100];this addWeaponCargo
    [""ACE_G3A3"", 10]; {this addMagazineCargo [_x, 20];} forEach [""ACE_DM25"",
    ""ACE_DM31"", ""ACE_DM33"",""ACE_DM51A1""]; this addWeaponCargo
    [""BWMod_PzF"", 4];  this addMagazineCargo [""BWMod_PzF_3"", 15]; this
    addMagazineCargo [""ACE_LargeBandage"", 20];   this addMagazineCargo
    [""ACE_Morphine"", 15];   this addMagazineCargo [""ACE_Medkit"", 10];   this
    addMagazineCargo [""ACE_Epinephrine"", 15];"
```

Eine Menge Code – aber keine Bange, das meiste davon ist redundant in dem Sinne, dass wir den Code für alle Fahrzeugtypen, die wir als Nachschub zulassen, wiederholen müssen. Beginnen wir.

Zeile 4 beginnt mit der bereits erwähnten Abfrage, welcher Spieler gerade das Skript ausführt. Da eine Anforderung an unser Skript war, nur bestimmten Personen den Zugriff auf das Nachschubmenü zu gewähren, erfüllen wir diese Forderung mit dieser Zeile, da alle Personen, die nicht `s1h` oder `s1z` sind, das Skript sofort wieder beenden. Diese Variablen stehen einfach für zwei beliebige Einheiten, bei uns war es der Kommandant/die OPZ sowie der Nachschuboffizier. Der Rest des Skriptes wird also nur von diesen beiden Spielern lokal ausgeführt. Da der Aktionsmenüeintrag immer lokal ist, sehen auch nur diese zwei an den beiden Fahrzeugen `wolf1` und `wolf2` die Einträge. Trotzdem haben wir noch eine zweite Sicherung eingebaut, die wir gleich besprechen werden.

Als nächstes erfolgt in den Zeilen 7 bis 9 die Initialisierung der wichtigen ARRAYS `respawn_array` und `respawn_array_ori`. Diese beinhalten zunächst die drei möglichen Respawn-Positionen. Dabei steht `respawn_vehicle_x` für einen der drei Variablennamen der unsichtbaren H-Objekte, die im Editor platziert wurden. Später können wir durch den Variablennamen auf das Objekt zugreifen und die Position abfragen. An dieser Stelle ist keine PUBLICVARIABLE notwendig! Da beide Spieler zur selben Zeit, nämlich zu Beginn der Mission, dieses Skript aufrufen, legen beide für sich lokal die gleiche Variable mit demselben Inhalt an, zusätzlicher Traffic wäre hier also nicht sinnvoll.

Hernach folgen die Zeilen 11 bis 17. Hierbei handelt es sich um eine FOREACH-Schleife für die beiden Fahrzeuge `wolf1` und `wolf2`, die alle benötigten Aktionseinträge bekommen sollen. Der ADDACTION-Befehl ist sehr mächtig. In seiner einfachsten Variante bedarf er lediglich zweier Argumente:

```
ID = player addAction ["Ein Auslöser!", "skript.sqf"];
```

Das erste Argument ist der Anzeigename im Aktionsmenü und kann mit den bekannten XML-Tags auch farblich verändert werden (`<t color='#ffffff'>Text</t>`). Möchte man den Eintrag später wieder entfernen, kann dies über den gespeicherten Rückgabewert geschehen:

```
player removeAction ID;
```

Dies geschieht mittels des Befehls **removeAction**.

Die wahre Macht des Befehls ADDACTION liegt aber in seiner weitaus größeren optionalen Parameterliste. Wir geben hier der Vollständigkeit halber alle Möglichkeiten noch einmal an, überlassen dem Leser aber die Erprobung aller Parameter:

```
ID = unit addAction [title, script, arguments, priority, showWindow,
hideOnUse, shortcut, condition, positionInModel, radius, radiusView,
showIn3D, available, textDefault, textToolTip]
```

Wenn wir uns oben noch einmal die Zeilen 11 bis 17 anschauen, sehen wir, dass wir in unserem Beispiel alle optionalen Parameter bis `condition` angegeben haben. Das ist leider ein kleines Übel, das wir eingehen müssen, wenn wir optionale Parameter benutzen möchten: Es müssen immer alle vorherigen, nicht aber die nachfolgenden Parameter definiert werden. Da wir den Parameter `condition` benutzen möchten, müssen wir alle Parameter davor ebenfalls angeben, damit der Compiler weiß, welche Parameter wir denn jetzt genau spezifiziert haben. Die `Condition` kann in einfache Anführungszeichen eingeschlossen werden, da der Compiler weiß, dass es sich hierbei um Code handelt. In unserem Beispiel haben wir noch einmal sichergestellt, dass der Eintrag nur von den Spielern `s1h` und `s1z` gesehen werden kann. Daher die Bedingung:

```
_this == s1h || _this == s1z[48]
```

Das wir `_this` in diesem Zusammenhang benutzen dürfen, ist eine weitere Besonderheit nur dieses Parameters. Es stehen zwei magische Variablen zur Verfügung: `_this` für den, der den Aktionsmenüeintrag auswählt und `_target` für das Objekt, das den Eintrag besitzt. Damit sollte alles Wesentliche zu diesen Zeilen gesagt sein. Wie bereits erwähnt ist die Benutzung des `condition`-Parameters hier redundant, da wir das gleiche bereits mit der if-Abfrage erreichen. Dennoch: mit EXITWITH bewirken wir eine Beendigung des gesamten Skriptes, also steht den übrigen Spielern z.B. auch die Variable `respawn_array` nicht zur Verfügung. Mit dem `condition`-Parameter hätten wir nur das Anzeigen des Aktionsmenüeintrages verhindern können, der Rest wäre dennoch von allen Spielern abgearbeitet worden.

Wir bitten den Leser besonders auf den Übergabeparameter zu achten! Dieser steht immer an dritter Stelle eines ADDACTION-Befehls. In unserem Falle ist es ein ARRAY mit nur einem Element, insofern hätten es auch runde Klammern getan, aber der Gewohnheit halber nutzen wir standardmäßig ein ARRAY. Wir übergeben als einzigen Parameter einen String, und zwar den Typ des Fahrzeuges, das wir später respawnen wollen. Das ist wichtig, da wir diesen Parameter im Hauptskript noch auswerten müssen.

Damit sind alle Aktionsmenüeinträge an den beiden Fahrzeugen angebracht und können ab sofort von den beiden berechtigten Spielern benutzt werden. Es fehlen für den späteren Respawn nur noch die Init-Zeilen, die als letztes in den Zeilen 20 bis 22 folgen.

An dieser Stelle vielleicht ein Wort der Erklärung: Obwohl wir nur 3 Plätze zur Auswahl haben, ist natürlich die Anzahl der Fahrzeugtypen unbegrenzt! Wir können ja auf diesen drei Plätzen alle erdenklichen Fahrzeuge respawnen lassen. In unserem Beispiel stehen als Nachschub zur Verfügung: der SPZ Marder, die Support-LKWs (Munition, Treibstoff sowie Reparatur) sowie ein großer 10t-LKW als Nachschublaster. Damit haben wir 5 Fahrzeugtypen, die jeweils eine vorgefertigte Init-Zeile erhalten. Diese speichern wir in aussagekräftige globale Variablen, da wir sie später ebenfalls im Hauptskript benötigen. Die Init-Zeilen bestehen im Wesentlichen aus den Befehlen **addWeaponCargo** und **addMagazineCargo**, da sie einfach Ausrüstung in den Stauraum des Fahrzeuges legen. Beide Befehle erwarten als Argument ein ARRAY, das sowohl den Klassennamen des Gegenstandes als auch die Anzahl enthält.

---

[48] Auch hier ganz große Vorsicht! SQF hasst nichts mehr als nicht vorhandene Variablen. Wenn ihr dieses Skript selbst ausprobiert und nur eine Einheit mit dem Namen `s1h` auf dem Editor platziert, wird das Skript nicht funktionieren, aber es kommt keine Fehlermeldung! Sehr häufiger Fehler: Alle Einheiten, die abgefragt werden, müssen auch existieren, sonst beendet SQF sofort das Skript. Wenn eine Einheit namens `s1z` nicht existiert, passiert einfach gar nix!

Wir erinnern an dieser Stelle noch einmal an den Grund, diesen zweiten Schritt in ein zweites Skript auszulagern, anstatt alles in das Hauptskript zu schreiben. Zum einen wird das Hauptskript bei jedem Aufruf über ein Aktionsmenü aufgerufen, es wäre also sinnlos, immer wieder die gleichen globalen Variablen neu zu belegen, das würde sogar zu Fehlern führen. Außerdem wäre es für die Wartung der Skripte sehr aufwändig, immer alle Einstellungen in einer sehr großen Datei vorzunehmen und neu herauszusuchen. Die Datei `vehicle_respawn_init.sqf` ist aber so angelegt, dass sie sämtliche wichtige Einstellungen enthält. Möchte jemand in seiner Mission statt der drei Respawn-Plätze vier oder fünf oder auch weniger, also zwei oder einen, benutzen, so braucht er lediglich die Variable `respawn_array` anpassen, der Rest funktioniert dann automatisch. Ebenso müssen die berechtigten Spieler nur in dieser Datei angepasst werden. Daher plädieren wir bei größeren Objekten immer dafür, ein Skript so zu gestalten, dass das Hauptskript möglichst unabhängig von Einstellungen funktioniert und eine zweite, kleinere Datei alle wichtigen Variablen und Einstellungen beinhaltet bzw. setzt, die für das Funktionieren notwendig sind.

Die Vehicle_respawn.sqf

Damit kommen wir zum dritten und letzten Teil dieses Beispiels, dem Hauptskript. Da dieses sehr umfangreich ist, präsentieren wir nur einen Teil des Codes und bitten euch, den Rest via Link selbst herunterzuladen. Danach solltet ihr das Skript mit einem Texteditor eurer Wahl öffnen, der euch Zeilennummern anzeigt, um den Erklärungen folgen zu können.

Example_Vehicle_vehicle_respawn.sqf

```
1  // Respawnscript by James
2
3  // Auslesen der übergebenen Parameter unit
4  _vehicle = _this select 0;
5  _unit = _this select 1;
6  _id = _this select 2;
7
8  // Übergebener Fahrzeugtyp, der spawnen soll!
9  // Speichert die aktuelle Zeit
10 _type = (_this select 3) select 0;
11 _time = diag_tickTime;
12
13 // Muss außerhalb von while deklariert werden
14 _respawn_pos = false;
15
16 // Ermittelt den freien Stellplatz! Scope-Fehler! Respawn ist nur innerhalb
     bekannt!
17 while {(typename _respawn_pos == "BOOL")} do
18 {
19   // Möglichkeit, Schleifen gezielt zu benennen und abzubrechen! exitWith
       bricht nur die aktuelle, innerste ab!
20   ScopeName "while_Schleife";
21   {
22     _umgebung = nearestObjects [getpos _x, ["Car","Tank"], 3];
23     if (count _umgebung == 0) then
24       {
25         _respawn_pos = _x;
```

Zunächst rufen wir uns noch einmal ins Gedächtnis, was das Hauptskript vollbringen soll (s. Graphik Seite 92). Das Skript wird vom Spieler mittels Aktionsmenü aufgerufen und bekommt den Fahrzeugtyp genannt, den es als Nachschub bereitstellen soll. Dabei wollen wir, dass der Nachschub auf einem der drei vorbestimmten Plätze geliefert wird und die Anfrage zudem nicht möglich sein soll, wenn alle Plätze belegt sind oder drei Anfragen gleichzeitig abgesetzt wurden.

Wir beginnen ganz klassisch in Zeile 4-6 mit dem Auslesen und Speichern der übergebenen Parameter, die der Befehl ADDACTION immer mit übergibt!. Tatsächlich brauchen wir im gesamten Skript keine der drei Variablen, es wäre aber denkbar, das Skript z.B. dahingehend abzuändern, dass ein Spieler nur einmal ein Fahrzeugtyp anfordern kann und dann warten muss. Dann wäre die dazu notwendige Zeile bereits soweit vorbereitet:

```
_vehicle removeAction _id;
```

Wir wollen aber, dass der Spieler tatsächlich auch drei Mal parallel das gleiche Fahrzeug bestellen kann. Daher haben wir keine solche Zeile.

Wichtig ist hingegen der optional übergebene Parameter, den wir bei diesem Befehl immer an 4. Stelle finden! Das heißt, ADDACTION gibt jedem Skript immer ein ARRAY von Übergabeparametern mit. Dabei sind die ersten 3 Elemente: das Objekt, das den Eintrag besitzt, die auslösende Einheit, die ihn aktiviert hat sowie die ID des Eintrages selber. An vierter Stelle kommen dann unsere eigenen Parameter, wie wir sie in der `vehicle_respawn_init.sqf` ja festgelegt hatten, nämlich jeweils einen STRING mit dem Fahrzeugtyp. Da wir auch bei einem einzigen Element immer ein ARRAY übergeben, haben wir daher ein 2D ARRAY und müssen in Zeile 10 daher auch einen zweifachen SELECT-Befehl benutzen.

Jetzt folgt die eigentliche Crux der ganzen Geschichte, die Positionsvergabe. Wir haben drei Positionen, gespeichert in der Variable `respawn_array`. Jetzt müssen wir uns überlegen, wie wir es schaffen, diese Positionen darauf zu überprüfen, ob ein Fahrzeug in der Nähe ist. Dies lösen wir mit dem Befehl **nearestObjects**. Dieser besitzt als Argument ein ARRAY aus drei Elementen: Das erste Element gibt die Suchposition an, das zweite Element ist ein Filter, der alle Klassennamen annehmen darf, die in der CFGVEHICLES definiert sind und das letzte Element gibt den Suchradius an, den der Befehl abgrasen soll. Damit haben wir alles, was wir brauchen. Daher ist der Kern des Skriptes die Zeile 22, in der wir genau diesen Befehl bzw. genauer gesagt dessen Rückgabewert, nämlich ebenfalls ein ARRAY, in der Variable `_umgebung` speichern. Ist dieses ARRAY nun leer, dann haben wir gewonnen, denn dann steht offensichtlich kein Fahrzeug auf dem Feld. Damit wir alle Positionen überprüfen können, sollte klar sein, dass wir das Ganze in eine FOREACH-Schleife packen (Zeilen 21-32). Wie bekommen wir jetzt die Position? Nun, sofern unser ARRAY leer ist, was gleichbedeutend damit ist, dass die Anzahl der enthaltenen Elemente 0 ist (Zeile 23), dann speichern wir das aktuelle Element der FOREACH-Schleife in der Hilfsvariable `_respawn_pos`. Jetzt müssen wir aber aufpassen: Wir müssen natürlich verhindern, dass diese Position bei einer zweiten Anfrage noch zur Verfügung steht. Ebenso müssen wir verhindern, dass ein zweiter Spieler diese Position noch zur Verfügung hat! Sonst würde Folgendes passieren: Das Skript stellt fest, dass Platz 1 frei ist und beginnt mit der Arbeit. Gleichzeitig sende ich aber eine zweite Anfrage, und ein zweiter Spieler eine dritte. Beide Anfragen finden ebenfalls, dass Platz 1 noch frei ist und stellen dort zusätzlich ihre Fahrzeuge ab. Im Ergebnis haben wir 3 Fahrzeuge auf einem Respawn-Platz. Das geht nicht. Daher nehmen wir direkt, nachdem wir festgestellt haben, dass eine Position frei ist, diese aus dem ARRAY der Positionen heraus (Zeile 28). Damit können wir selbst diesen Platz nicht mehr nutzen, denn jedes Skript, das wir jetzt noch aufrufen, greift ja ebenfalls auf die Variable `respawn_array` zu. Diese ist aber für uns global und damit zumindest auf meinem Client immer gültig. Aber ein anderer Spieler kann immer noch auf die eben genannte Position zugreifen. Um dies zu verhindern, müssen wir die Variable `respawn_array` von einer globalen in eine öffentliche, also PUBLICVARIABLE umwandeln. Da dies aber leider nicht dauerhaft geht, müssen wir immer mit dem Befehl PUBLICVARIABLE sozusagen eine Änderung an alle Spieler übertragen. Dies geschieht in Zeile 29. Damit ist sichergestellt, dass alle Spieler die geänderte Variable erhalten, was gleichbedeutend damit ist, dass die aktuelle Position `_x` aus allen Variablen bei allen Spielern entfernt wurde. Damit wären wir eigentlich fertig. Aber leider nur eigentlich. Das richtig Schwierige an der ganzen Angelegenheit ist jetzt die Lösung des Problems, den Vorgang abzubrechen, wenn kein Stellplatz gefunden wird! Denn dann soll

der Spieler ja darüber informiert werden, dass momentan alle Plätze belegt sind. Nur wie stellen wir das an? Unsere FOREACH-Schleife läuft über alle vorhandenen Positionen und wir prüfen auf eventuell vorhandene Objekte in deren Nähe. Was soll jetzt aber passieren, wenn kein Platz gefunden wird, weil immer ein Fahrzeug vorhanden ist? Meine Lösung ist eventuell etwas aufwändig und kompliziert, aber sie verdeutlicht noch einmal den Nutzen von SCOPENAME.

Zunächst wird nämlich die Variable `_respawn_pos` als BOOLEAN definiert. Wir initialisieren sie mit dem Startwert FALSE, damit wir wissen, dass zu diesem Zeitpunkt noch keine Position feststeht. Wichtig ist auch, dass wir die Zeit in der Variable `_time` festhalten, damit wir feststellen können, wann 10 Sekunden vergangen sind. Wir möchten nämlich, dass die Anfrage 10 Sekunden lang geprüft wird. Andernfalls bräuchten wir überhaupt keine WHILE-Schleife! Denn wenn keine Position frei wäre, würde der FOREACH-Block einfach abgearbeitet, ohne ein Ergebnis zu erzielen und wir könnten die ganze umschließende WHILE-Schleife einfach löschen. Wir möchten aber den Zeitaspekt berücksichtigen, daher brauchen wir sie.

Wir lautet nun die Ausführungsbedingung für die WHILE-Schleife? Offenbar wollen wir die Schleife solange ausführen, bis wir eine Position gefunden haben. Das geschieht, indem wir der Variable `_respawn_pos` einen Inhalt geben. Wir hätten also zum einen die Möglichkeit, die Existenz der Variable `_respawn_pos` zu prüfen und zwar mit ISNIL. Da wir diese Möglichkeit aber schon kennen, wollen wir das ganze anders lösen. Wir haben uns ja etwas dabei gedacht, die Variable zunächst als Boolean anzulegen, denn was passiert, sobald `_x` in `_respawn_pos` gespeichert wird? Es ändert sich der Datentyp! Und zwar von BOOLEAN zu OBJECT. Und genau das können wir mit dem Befehl **typeName** prüfen, dieser bestimmt nämlich den Datentyp einer Variable. Daher lautet sowohl die Bedingung in der while-Schleife als auch später in der if-Abfrage:

```
typename _respawn_pos == "BOOL"
```

Dies ist eben genau solange der Fall, wie `_respawn_pos` nicht mit einer Position belegt wird. Jetzt haben wir aber noch ein Problem: Für den Fall, dass eine Position frei ist, wir also `_x` in `_respawn_pos` speichern, wollen wir ja bitte die Schleife verlassen. Wie können wir dies machen? Natürlich könnten wir einfach die Zeit abwarten, bis die Schleife von alleine ihren Kopf erneut prüft, aber wir haben bereits gelernt, dass wir jede Schleife durch SCOPENAME verlassen können. Daher führen wir im Inneren der FOREACH-Schleife den Befehl BREAKOUT durch (Zeile 30), um die Schleife sofort gänzlich zu verlassen. Wir erinnern uns, dass exitWith nicht gereicht hätte, da dies uns lediglich aus der FOREACH-Schleife, nicht aber der WHILE-Schleife, wirft.

Damit kommen wir zu Zeile 40, in der wir prüfen, ob nun eine Anfrage positiv beschieden wird oder nicht. Wenn bis dato keine freie Position gefunden wurde, dann muss _respawn_pos noch immer ein BOOLEAN vom Wert FALSE sein, also bricht das Skript ab und teilt dem Spieler die traurige Nachricht mit. Andernfalls passiert gar nix und es geht weiter mit Zeile 43.

Jetzt folgt lediglich eine große Mehrfachverzweigung, die einzig und alleine davon abhängt, was der Spieler für einen Aktionsmenüeintrag ausgewählt hat, da wir jetzt den optionalen Parameter `_type` abfragen. Der Code ist für alle Teile identisch, daher beschränke ich mich auf einen Block.

Zunächst wird das Fahrzeug nach einer individuell einstellbaren Zeit gespawnt. Dies geschieht in Zeile 52 mit dem CREATEVEHICLE-Befehl. Als Position kommt natürlich die Variable `_respawn_pos` zum Einsatz, die wir uns ja mühsam erarbeitet haben.

Nun müssen wir dafür sorgen, dass das Fahrzeug seine init-Zeile bekommt und ausführt. Problematisch dabei ist wie immer, dass der Befehl lokal ist, wir aber natürlich eine globale Wirkung wollen, da alle Spieler den gleichen Inhalt im Fahrzeug sehen sollen. Daher brauchen wir an dieser Stelle den Public-EventHandler, den wir in der `Init.sqf` eingangs formuliert hatten.

Der Public-EH springt immer an, wenn die Variable `global_exe` verändert und übertragen wird. Da wir einen Code ausführen wollen, speichern wir in dieser Variable als STRING unseren Code. Dies geschieht in

den Zeilen 55-60. Mit dem Befehl **setVehicleInit**[49] können wir die vordefinierte Init-Zeile aus der `vehicle_respawn_init.sqf` zuweisen und anschließend mit dem Befehl processInitCommands ausführen. Der Befehl setDir dient dann lediglich zur korrekten Ausrichtung.

Damit der Public-EH auch anspringt, müssen wir noch einmal den Code an alle Clients mit PUBLICVARIABLE übertragen.

Dabei muss man immer beachten, dass ein Public-EH niemals auf dem Client ausgeführt wird, der selbst die Variable überträgt. Daher müssen wir nun als letztes den Code noch einmal beim Spieler selbst ausführen, der auch das Skript ausführt. Dies geschieht in Zeile 66.

Bis auf die Meldungen, die jeder nach Gutdünken gestalten kann, war es das. Alle anderen Cases folgen dem gleichen Aufbau.

Wichtig ist aber die allerletzte Zeile, denn wir dürfen nicht vergessen, dass wir ja eine Position gestrichen haben. Wenn das Fahrzeug aber schließlich geliefert wurde, müssen wir die Position wieder freigeben, da sonst nur insgesamt drei Mal überhaupt Nachschub bestellt werden könnte. Daher müssen wir einfach ganz zum Schluss die Position wieder in das ARRAY `respawn_array` hinzufügen und erneut mit PUBLICVARIABLE übertragen.

Ich hoffe, das Beispiel ermuntert zum eigenen Ausprobieren, war verständlich und hilfreich und hat euch einige wichtige Konzepte noch einmal in der Praxis verdeutlichen können.

[49] Wie man nachlesen kann, wurde dieser Befehl in ArmA3 deaktiviert, stattdessen soll BIS_fnc_MP benutzt werden.

III. 3. BORDFUNK IN FAHRZEUGEN

Mal wieder ein kleines Beispiel aus meiner eigenen Ideen-Schmiede. Sicherlich wird es hierfür zahlreiche – auch bessere – Umsetzungen geben, aber ich finde das Beispiel gerade wegen seiner Schlichtheit erwähnenswert.

Zunächst die Idee: In vielen Missionen ist es vom Missionsbauer gewünscht, dass der einfache Soldat über keine technischen Spielzeuge (manchmal auch Schnickschnack genannt…) verfügen soll. Dazu gehören am häufigsten das GPS und das Funkgerät (wir gehen natürlich davon aus, dass ACRE ein fester Bestandteil eures Modstrings ist). Auf der anderen Seite sind viele Missionen aber als Missionen angelegt, bei dem abgesessene und aufgesessene Kräfte zusammen operieren sollen. So ist es z.B. auch beim SPZ Marder oder TPZ Fuchs, die beide bei uns zum Einsatz kommen. Hier hat normalerweise der Fahrzeugkommandant (SPZ Marder) den Oberbefehl über das Fahrzeug sowie die abgesessenen Truppen, die von einem Truppenführer befehligt werden. Nun hätte der Kommandant als Gruppenführer seiner Einheit aber als einziges ein Funkgerät, als höchstes der Gefühle eventuell noch sein Stellvertreter. Die einfachen Soldaten, und damit auch die Besatzung des SPZ sowie die hintere Kampfraummannschaft, hätte aber keine Möglichkeit, über Funk zu kommunizieren. Gerade für Manöver und im Einsatz ist es aber fast unmöglich für die Besatzung, die Kommunikation nur via Stimme zu gewährleisten, dafür ist der Lärm einfach zu stark. Hier wollen wir jetzt ein Skript schreiben, dass es den Spielern ermöglicht, über eine Bordsprechanlage miteinander zu kommunizieren, obwohl sie eigentlich keinerlei Funkgeräte am Mann haben.

Dazu müssen wir uns sicherlich irgendwie ein ACRE-Funkgerät besorgen. Wir wollen natürlich das Addon des BW-Mods nicht anpassen und irgendwie ein Funkgerät in das Fahrzeug integrieren, dazu fehlen uns sowohl die Zeit als auch das Wissen. Wir gehen stattdessen einen kleinen Umweg und lösen das Problem im Grunde sehr einfach und elegant: Wir geben dem Spieler nur ein Funkgerät, wenn er sich **im** Fahrzeug befindet und entfernen es wieder, sobald er aussteigt. Damit hätte er im Fahrzeug die Möglichkeit, über das kleine Funkgerät 343 mit der Besatzung zu reden, hätte aber außerhalb des Fahrzeuges nach wie vor kein Funkgerät, so wie es der Missionsbauer wollte.

Da wir jetzt die Aufgabe vor Aufgabe vor Augen haben, fehlen uns nur noch die Zutaten. Das ganze können wir problemlos zu Beginn der Mission irgendwie verankern, das Skript setzen wir als normale SQF-Datei auf und rufen es wahrscheinlich zu Beginn der Mission auf, da wir ja permanent sozusagen die Bordfunkanlage nutzen wollen. Wir müssen also im Grunde nur feststellen, wann ein Spieler ein Fahrzeug betritt und ihm dann ein Funkgerät geben, sowie es wieder entfernen, sobald er das Fahrzeug verlässt.

Wer an dieser Stelle gut aufpasst und mitdenkt, wird eine zunächst naheliegende Lösung mittels WHILE-Schleife und permanenter Zustandsprüfung des Spielers sicherlich nicht favorisieren. Natürlich wäre es möglich, von Beginn an für alle Spieler permanent zu prüfen, ob der Spieler gerade in einem Fahrzeug sitzt, aber das wäre nicht nur sehr performance-lastig, es wäre auch schlicht und einfach nicht elegant. Viel besser ist dagegen die Erkenntnis, dass es sich bei unserem Problem um ein ganz klares Event, also Ereignis handelt, wir erinnern an Kapitel I. 8. Wenn wir einzig und allein das Ereignis des Einsteigens in ein Fahrzeug brauchen, ja warum schauen wir dann nicht direkt bei EH nach, ob es da etwas für uns gibt? Genau das tun wir auch und finden folgenden EH:

GetIn

Triggered when a unit enters the object (only useful when the event handler is assigned to a vehicle). It does not trigger upon a change of positions within the same vehicle. It also is not triggered by the moveInX commands.

Global. EG Global

Passed array: **[vehicle, position, unit]**

- vehicle: Object - Vehicle the event handler is assigned to
- position: String - Can be either "driver", "gunner", "commander" or "cargo"
- unit: Object - Unit that entered the vehicle

Was will man mehr im ArmA-Leben! Der EH feuert genau dann, wenn eine Einheit ein Objekt (hier Fahrzeug) betritt und liefert als Übergabeparameter an ein eventuelles Skript das Fahrzeug, die Sitzposition sowie die eingestiegene Einheit. Na wenn sich damit nicht arbeiten lässt.

Damit können wir zunächst den Aufruf unseres Skriptes vorbereiten. Ein EH muss immer an dem Objekt „angebracht" werden, an dem es auf ein Event „lauschen" soll. In unserem Falle ist also das Ziel nicht etwa eine Einheit, sondern natürlich das Fahrzeug, wie der Name GETIN ja auch unschwer erkennen lässt. Wir setzen also neben einer Spieler-Einheit zum Testen einen SPZ-Marder (oder ein Fahrzeug eurer Wahl) auf die Karte des Editors. Danach schreiben wir in die Init-Zeile des Fahrzeuges den folgenden Code:

```
this addEventHandler ["GetIn", {[_this select 0, _this select 2] execVM
"scripts\bordfunk.sqf";}];
```

Mit `this` nehmen wir Bezug auf das Fahrzeug und sparen uns das Abtippen des Variablennamens. Danach folgt der EH über den bekannten Befehl `addEventHandler`. Dieser besteht immer aus zwei Argumenten: Dem Namen des EH und dem auszuführenden Code-Teil im Falle des „Feuerns". Innerhalb des Code-Blockes ({…}) starten wir unser Script mit dem Namen „`bordfunk.sqf`", das im Ordner `scripts` liegt. Jetzt überlegen wir uns die Parameter, die wir dem Skript mitteilen wollen. Ganz klar werden wir das Fahrzeug brauchen, das den EH ausgelöst hat. Da der EH ja selbst ein ARRAY übergibt, sehen wir, dass das Fahrzeug als **vehicle** das 1. Element ist, das übergeben wird. Daher übergeben wir unserem Skript zunächst als 1. Übergabeparameter `_this select 0`, wobei `_this` für das ARRAY steht, dass der EH selbst erzeugt und übergibt, `select 0` liest dann gerade **vehicle** aus. In meinem Skript habe ich die **position** nicht mit übergeben, da ich möchte, dass alle Soldaten im Fahrzeug über Bordfunk verfügen. Hier wäre es also möglich, mein Skript anzupassen und auch **position** zu übergeben. Damit wäre es dann ferner möglich, z.B. nur der Besatzung (`driver`, `gunner`, `commander`) ein Funkgerät zuzuweisen, nicht aber dem hinteren Kampfraum (`cargo`). Da unser Skript ein Funkgerät an den Spieler austeilen soll, brauchen wir aber als letzten Übergabeparameter noch die Einheit, die in das Fahrzeug eingestiegen ist. Das wäre dann **unit**, das 3. Element im Array des EH und unser 2. Übergabeparameter, daher übergeben wir noch `_this select 2`. Das war's. Wenn wir jetzt in das Fahrzeug einsteigen, würde unser Skript starten, sofern wir es bereits im entsprechenden Ordner gespeichert haben.

Daher möchte ich jetzt zunächst mein Script vorstellen und euch die Zeit geben, euch einzulesen und mit der Funktionsweise vertraut zu machen. Anschließend gehen wir natürlich auf die Feinheiten ein, ich erkläre kurz die Funktionsweise und warum einige Dinge notwendig waren, obwohl sie zunächst überflüssig erscheinen. Da der gesamte Code aber ausführlich kommentiert ist, klärt sich das meiste eigentlich bereits beim Lesen.

bordfunk.sqf

```
1   // Bordfunkskript by James
2
3   // Auslesen der Parameter
4   _vehicle = _this select 0;
5   _unit = _this select 1;
6
7   // da im MP der Event-Handler bei ALLEN auslöst, müssen wir hier das Script
    stoppen!
8   if (player != _unit) exitWith{};
9
10  // Beendet das Skript an dieser Stelle, wenn das Fahrzeug nicht ein SPZ
    Marder ist
11  // wird nicht gebraucht, wenn das Script über einen Event-handler in der
    init des Fahrzeugs gestartet wird!
12  // if (typeof _vehicle != "BWMod_Marder_1A5") exitWith{};
13
14  // Bordfunk alias 343 wird dem Spieler hinzugefügt
15  _unit addWeapon "ACRE_PRC343";
16
17  // notwendig, damit ACRE das Funkgerät korrekt umbenennt, ohne Sleep geht
    das Skript nicht!
18  sleep 1;
19
20  // Da das Funkgerät eine ID bekommt, sobald man es aufnimmt, müssen wir den
    richtigen Namen erst finden
21  // Dies funktioniert nur, wenn KEINE Waffe aufgenommen wird, nachdem man das
    Fahrzeug betritt.
22  // Da es direkt nach dem Einstieg abgefragt wird, kann man auch noch eine
    PzF 3 oder sonstwas aufnhemen!
23  _funkgeraet = weapons _unit select ((count weapons _unit) -1);
24
25  hintSilent "Bordfunk verfügbar!";
26
27  // Solange der Spieler im Fahrzeug ist, passiert nichts weiter
28  waitUntil {(vehicle _unit) == _unit};
29
30  // Erst wenn er aussteigt, wird das Funkgerät entfernt.
31  _unit removeWeapon _funkgeraet;
```

Das ist also das ganze Skript. Hätte ich alle Kommentare weggelassen, hätten wir lediglich einen 9-Zeiler! Also umfangreich ist dieses Beispiel nicht. Gerade deshalb eignet es sich aber so gut für dieses Handbuch, weil dennoch wichtige Konzepte rekapituliert werden. Zunächst beginnen wir immer mit der Abfrage der übergebenen Parameter. Prinzipiell wäre dies nicht nötig, da wir überall, wo wir diese brauchen, auch direkt mit einer Abfrage wie `_this select 0` arbeiten könnten. Nur wofür steht das? Was wäre, wenn ihr euer Skript einem Freund aushändigt, der es ebenfalls benutzen, aber auch erweitern möchte? Ihr hättet keine Ahnung, wofür `_this select 0` steht, außer ihr schaut im EH nach! Das ist aber mehr als umständlich, also tun wir uns und ihm einen Gefallen und speichern das Ganze in einer ordentlich benannten Hilfsvariable! Da wir lediglich zwei Übergabeparameter benutzt haben, brauchen wir auch nur zwei Hilfsvariablen, in meinem Fall sind das `_vehicle` und `_unit`.

Nun ist mir beim Testen eine Besonderheit passiert, als ich Zeile 8 noch nicht hatte: Das Skript hat offenbar zu oft ausgelöst! Ich hatte auf einmal immer 2 Funkgeräte in meiner Ausrüstung und hatte überhaupt keine Idee, woher das kam. Ich hatte aber auch nur mit einem Kameraden das Skript getestet, daher wäre es mir früher aufgefallen, wenn ich mit zehn getestet hätte, denn dann hätte ich auch zehn Funkgeräte gehabt…

Die Lösung liegt in der Funktionsweise des EH. Dieser ist in BI Wiki als global markiert und arbeitet demzufolge auch global. Was dies bedeutet, wird uns jetzt schmerzlich bewusst: Der EH feuert auf **allen** Clients und dem Server. Auch wenn wir die Person sind, die in das Fahrzeug einsteigt, wird doch der EH und damit unser Skript bordfunk.sqf auf allen Clients aktiviert und damit ausgelöst. Dann ist klar, dass auch die Zeile

```
_unit addWeapon "ACRE_PRC343";
```

für alle ausgelöst wird. Da wir aber nicht PLAYER sondern `_unit` benutzt haben, wird bei jedem Client mit dieser Zeile mir als der einsteigenden Einheit ein Funkgerät zugewiesen! Ganz schön doof.

Die Lösung ist also eine vorzeitige Beendigung des Skriptes für alle Clients außer dem Spieler, der tatsächlich eingestiegen ist. Genau dafür steht uns der Befehl PLAYER zur Verfügung. Wir nutzen also unser Wissen aus Kapitel I. 9 „Vorzeitiges Beenden von Schleifen und Skripten – Sprungstellen“ und beenden das Skript mittels EXITWITH.

```
if (player != _unit) exitWith{};
```

Danach passiert erst einmal nichts Spannendes. Wir geben der Einheit ihr Funkgerät – den Klassennamen haben wir nachgeschaut oder über die Ace-Debugkonsole ermittelt (**weapons**). Anschließend müssen wir aufgrund der internen Arbeitsweise von ACRE eine kleine Pause einbauen, damit das Funkgerät korrekt von ACRE initialisiert wird (SLEEP-Befehl).

Das war's im Grunde dann auch schon. Damit steht dem Spieler ein Bordfunk zur Verfügung, jetzt kann er mit dem 343er auf einem beliebigen Kanal mit allen weiteren Insassen reden. Leider funktioniert das Ganze auch außerhalb des Fahrzeuges, denn das 343er hat ja eine Reichweite von mehreren 100 Metern, aber das soll uns erst einmal nicht stören, unsere Lösung funktioniert soweit.

Jetzt müssen wir uns aber auch Gedanken über das Entfernen des Funkgerätes machen. Das wäre sehr einfach, wenn ACRE nicht eine kleine Gemeinheit eingebaut hätte. Prinzipiell wäre der einzige Befehl, den wir bräuchten:

```
_unit removeWeapon "ACRE_PRC343";
```

Meisten ist es auch so einfach – nur wie gesagt: bei ACRE eben nicht. ACRE kennt zwar den Klassennamen "ACRE_PRC343" und gibt dem Spieler auch ein Funkgerät, danach wird dieses Funkgerät aber von ACRE mit einer Nummer versehen – das ist auch der Grund dafür, dass wir mehr als ein Funkgerät desselben Typs aufnehmen können, mit unterschiedlichen Frequenzen. Also müssen wir uns abermals eines Tricks bedienen, um das Funkgerät doch irgendwie zu speichern. Meine Lösung: Das ARRAY, das der Befehl WEAPONS liefert, wird natürlich zur Laufzeit des Spiels ergänzt, wenn der Spieler Waffen aufnimmt. Demzufolge befinden sich „Waffen“, die als letztes aufgenommen wurden auch am – ratet mal! – Ende, also als letzte Einträge. Wenn wir dem Spieler folglich ein Funkgerät zuweisen, muss dieses am Ende auftauchen. Und wie lesen wir ein Element am Ende eines ARRAYs dynamisch aus, wenn wir dessen Länge nicht kennen? Das Kapitel „Array“ lässt grüßen: Mit dem COUNT-Befehl!

```
_funkgeraet = weapons _unit select ((count weapons _unit) -1);
```

Damit ist das Funkgerät fest gespeichert, denn wir haben den Klassennamen in der Hilfsvariable fest zwischengespeichert.

Der Rest ist Standardkost: Eine Ausgabe für den Spieler, damit er auch weiß, was er Neues zur Verfügung hat, die Abfrage mit WAITUNTIL, ob der Spieler noch im Fahrzeug ist, und anschließend das Entfernen des Funkgerätes mit eben jenem Befehl **removeWeapon**.

Damit sind wir mit diesem Beispiel fertig. Das Skript funktioniert und ist MP-tauglich. In Zeile 12 seht ihr außerdem eine Möglichkeit, das Skript so anzupassen, dass nur bestimmte Fahrzeugklassen zugelassen sind, der Spieler als kein Bordfunk im Traktor erhält…

```
if (typeof _vehicle != "BWMod_Marder_1A5") exitWith{};
```

III. 4. GEGNER ANHALTEN ODER ZERSTÖREN/UNSCHÄDLICH MACHEN

Anhang

A. 1. LÖSUNGEN ZU DEN AUFGABEN

Kapitel I. 3

[1] Die drei Möglichkeiten waren der Aufruf direkt im Editor in der Initzeile eines Objektes oder in einem der drei Befehlszeilen eines Auslösers. Als dritte Möglichkeit kann ein externes Script geladen werden.

[2] RANDOM ist ein Befehl, der Zufallszahlen erzeugt. Dabei generiert er alle Zahlen zwischen einschließlich 0 und ausschließlich der angegebenen Obergrenze. D.h. dass RANDOM 1 z.B. alle Zahlen im Intervall $I = [0,1[$ erzeugt.

[3] RANDOM ist eine sehr sinnvolle Funktion, wenn wir Zufallswerte generieren wollen. Allerdings brauchen wir oftmals nur ganzzahlige Werte und dann wollen wir natürlich auch die Kontrolle über die Werte haben. Wenn wir also zufällig eine 0 oder 1 erzeugen wollen, ist `random 1` nicht sehr hilfreich, da es alle Werte zwischen diesen beiden und ab und zu auch die 0 selbst, nie aber die 1 erzeugt. Um dies zu lösen, müssten wir sinnvoll runden können. Genau dafür stehen uns drei Befehle in SQF zur Verfügung: **ceil** (rundet auf die nächstgrößere ganze Zahl), **floor** (rundet auf die nächstkleinere ganze Zahl) und **round** (rundet nach den bekannten mathematischen Regeln).
`ceil (random 1)` würde uns permanent 1, `floor (random 1)` dagegen permanent 0 liefern. Wir brauchen tatsächlich `round random 1`, damit immer nur entweder 0 oder 1 ausgegeben wird. Jetzt erkennt der ein oder andere vielleicht eine Möglichkeit, zufällige Elemente eines ARRAYs auszulesen.

Kapitel I. 4

[1] Die Aufgabe soll uns lediglich den Wert des Operators Module (%) verdeutlichen, auch wenn für Schaltjahre noch weitere Bedingungen gelten. Die Division durch 4 ist prinzipiell mit JAHR/4 möglich, liefert aber in allen Fällen außer den Schaltjahren einen Rest. Demnach kann die Lösung wie folgt formuliert werden. Mit einer Kontrollstruktur könnten wir dann natürlich weiter arbeiten:

```
Jahr % 4 == 0; // Der Rest der Division muss 0 sein
```

[2] Wenn wir irgendwelche Variablen auswerten wollen, brauchen wir immer Vergleichsoperatoren. In diesem Falle ist quasi die Gleichheit zwischen dem Spieler und einer Einheit gesucht. **Player** ist immer die Einheit des Spielers. Sofern wir im Editor alle Einheiten von Relevanz mit Variablennamen (Feld Name) versehen haben, können wir die Gleichheit von Spieler und Einheit mit „==“ prüfen. Demzufolge würde `player == s1` prüfen, ob der Spieler die Einheit s1 kontrolliert. Wollen wir das Gegenteil prüfen, negieren wir „==“ und benutzen „!=“: `player != s1`. Was uns jetzt noch fehlt, ist die Bedingung dafür, dass der Spieler eine Westeinheit ist. Dafür nutzen wir den Befehl SIDE. Dieser liefert den Rückgabedatentyp SIDE. Wie man nachlesen kann, wird `side player` also `west` liefern, wenn wir BLUEFOR spielen. Eine Verbindung von mehreren Bedingungen geschieht über die logischen Operatoren „&&“ sowie „||“. Daher lautet die Lösung:

```
(player == s1) && (side player == west);
```[50]

[50] Hier wird sehr gerne (auch von mir noch) west in „“ gesetzt. Side ist aber ein eigener Datentyp, kein String!

[3] Unser Beispielarray war [[1, 2], [3, 4]]. Wir hatten bereits verraten, dass wir mit `array select 1` auf das erste innere ARRAY [1, 2] zugreifen können. Von hier ist der Weg nicht mehr weit, denn wir haben es einfach mit einem erneuten ARRAY zu tun, über das wir wieder mit SELECT zugreifen können. Die Lösung lautet daher z.B.:

```
(array select 1) select 0; // liefert 1
```

Zur besseren Lesbarkeit wurden Klammern um den ersten auszuwertenden Ausdruck gesetzt.

[4] Hier gibt es verschiedene Möglichkeiten. Wir müssen zum Teil gegebene Werte durch neue ersetzen, zum anderen aber auch das ARRAY um ein Element erweitern. Wir präsentieren eine Möglichkeit[51]:

```
array1 set [0,1]; array1 set [1,false]; array1 set[3,4];
```

[5] Wir wissen, dass wir ein Element mit dem Befehl SELECT auslesen können. Alles, was wir dafür brauchen, ist die Index-Nummer des letzten Eintrages. Da wir die Länge des ARRAYs nicht kennen, müssen wir uns mit einem Zähl-Befehl behelfen, der für uns die Länge ermittelt. Anschließend brauchen wir nur noch zu berücksichtigen, dass das 1. Element immer mit 0 indiziert ist und nicht mit eins, daher muss die Länge um eins reduziert werden:

```
array select (count array - 1);
```

Kapitel I. 5

[1] Gefragt war, was folgender Code tut und warum dies funktioniert:

```
{player addMagazine "30Rnd_556x45_Stanag"} forEach [1,2,3,4];
```

Wir erinnern uns an die Funktionsweise der FOREACH-Schleife: Für jedes Element des Arrays wird der Code innerhalb des Blockes ausgeführt. Normalerweise würden wir diese Schleife im Zusammenhang mit `_x` nutzen, um für jedes Element eine spezifische Aktion auszuführen. Hier haben wir aber ein ganz anderes anliegen: Wir wollen einfach ein und dieselbe Aktion vier Mal hintereinander ausführen! In der Tat müssten wir ohne diese Lösung den Befehl einfach vier Mal kopieren. Mit FOREACH wird der Befehl aber sooft ausgeführt, wie es Elemente im ARRAY gibt. Und der Witz an der Sache ist, dass die Nummern überhaupt keine Rolle spielen! Denkt an die alternative Syntax einer FOREACH-Schleife als eine FOR-Schleife von `_i=0` bis Anzahl der Elemente im ARRAY - 1. Solange wir also das Element selbst mit `_x` nicht abfragen bzw. darauf zugreifen, bedeutet die obige Syntax nur, dass der Code vier Mal ausgeführt wird. D.h. es ist insbesondere genau dasselbe wie folgender Code:

```
{player addMagazine "30Rnd_556x45_Stanag"} forEach [1,1,"test",player];
```

[2] Wir wollen in dieser Übung bewusst nicht den Befehl VEHICLES benutzen, denn dieser birgt die beschriebene Problematik, der wir uns in der nächsten Aufgabe widmen. allUnits liefert uns zunächst alle Einheiten sowohl in als auch außerhalb von Fahrzeugen. `count allUnits` ist also nur die halbe Miete, wir müssen die Zählung weiter einschränken. Was wir brauchen, ist eine Unterscheidung zwischen dem Fahrzeug der Einheit und der Einheit selber. Dafür gibt es den Befehl VEHICLE. `Vehicle unit` liefert nämlich den Namen des Fahrzeugs (als Object) oder – falls die Einheit in keinem Fahrzeug ist – die Einheit selber! Das Beispiel auf BI Wiki ist genau das, was wir brauchen:

```
{(vehicle _x) != _x} count allUnits;
```

Damit zählen wir nur die Einheiten in ALLUNITS, die in einem Fahrzeug sitzen.

[51] Selbstverständlich hätte die 4 auch durch ein `Array1 + [4]` erreicht werden können. Wie bereits angedeutet, würden wir damit aber technisch ein neues Array erzeugen, daher ist der SET-Befehl die 100% korrekte Variante.

[3] Natürlich wäre es viel angenehmer, den vorgefertigten Befehl VEHICLES benutzen zu können, doch hier müssen wir etwas tricksen. Als Hinweis war bereits gegeben, dass wir die Benzinkanister quasi über ihren Typ- (oder Klasse-)namen rauswerfen. Die Frage ist natürlich, wie man so einen Klassennamen herausbekommt. Im Teil II – SQF im Multiplayer werden wir euch den SIX Config Browser vorstellen, und im Teil III – SQF in der Praxis werden wir euch noch weitere Methoden präsentieren, direkt im Editor einen Klassennamen herauszufinden. In unserem Falle können wir ausnutzen, dass wir zumindest das Objekt bereits haben, denn es taucht ja im ARRAY VEHICLES auf. Daher können wir z.B. das letzte Element, sofern es sich dabei um einen Benzinkanister handelt, mit SELECT abfragen:

```
typeOf (vehicles select (count vehicles -1));
```

Dies liefert uns den Klassennamen „ACE_JerryCan_15“. Jetzt muss man nur noch wissen, welchen Datentyp TYPEOF zurückgibt. In BI Wiki lesen wir nach: Es ist der Datentyp STRING. Das heißt, wir wollen alle Fahrzeuge in VEHICLES zählen, die nicht vom Typ „ACE_JerryCan_15“ sind:

```
{(typeOf _x) != "ACE_JerryCan_15"} count vehicles;
```

[4] Wir folgen dem Hinweis und betrachten den Befehl DISTANCE. Dieser berechnet die direkte Entfernung zweier Objekte oder Positionen und gibt sie als NUMBER zurück. Unser Spieler ist `player` und unser Fahrzeug nennen wir mal wieder sehr kreativ `auto`. Dann ist der Abstand zwischen beiden:

```
player distance auto;
```

Um eine Explosion zu erzeugen, kennen wir aktuell noch keinen Befehl. Wir behelfen uns an dieser Stelle mit dem Trick, dass ein Fahrzeug fast immer explodiert, wenn wir den Schaden auf 100%, also 1 setzen. Dazu kennen wir bereits den Befehl SETDAMAGE. Demnach müssen wir nur noch beides zusammenbringen:

Loesung_1.sqf

```
1  _grenzabstand = 15;
2  if (player distance auto < _grenzabstand) then
3  {
4    auto setDamage 1
5  }
```

Ohne ACE und das Modul Munitionsbrand ist der Effekt aber sehr bescheiden und beschränkt sich auf eine Flamme. Wir werden in Teil III – SQF in der Praxis zeigen, wie man bessere Explosionen erzeugt.

[5] Bisher haben wir den Spieler als „Zünder“ benutzt bzw. als Referenzobjekt für die Entfernungsbestimmung. Möchten wir nun, dass das Fahrzeug wirklich bei jedweder Person zündet, können wir nicht mehr mit einzelnen Einheiten arbeiten, denn dann müsste quasi für jede Einheit ein solches Skript parallel laufen. Wir behelfen uns daher mit dem Befehl ALLUNITS, dafür ist es ja schließlich da. Die Idee ist: Sobald ein Element aus ALLUNITS dem Fahrzeug näher als 15 Meter kommt – kaboom!

Loesung_2.sqf

```
1  _grenzabstand = 15;
2  {
3    if (_x distance auto < _grenzabstand) then
4    {
5      auto setDamage 1
6    }
7  } forEach allUnits
```

[6] Für den ersten Teil der Aufgabe müssen wir nur unsere bisherige if-Abfrage um eine Schadensabfrage ergänzen:

Loesung_3a.sqf
```
1  _grenzabstand = 15;
2  {
3    if (_x distance auto < _grenzabstand && damage auto > 0) then
4    {
5      auto setDamage 1
6    }
7  } forEach allUnits
```

Eine Meldung bewerkstelligen wir gewohnt mit dem HINT-Befehl. Da wir den Spieler konkret über seinen Status informieren wollen, geben wir auch den Schaden in Prozent mit aus:

Loesung_3b.sqf
```
1  … // Code von oben bis einschließlich Zeile 7
2  sleep 2; //Zeit zwischen Explosion und eigenem Status
3  if (damage player > 0) then
4  {
5    _schaden = damage player * 100;
6    hint ("Sie wurden verletzt! Sie haben "+str(_schaden)+
     " Prozent Schaden erlitten.")
7  }
8  else { hint "Sie wurden zum Glück nicht getroffen!" }
```

Ok, nicht verzweifeln. Das gemeine an diesem Beispiel ist, dass es eine Menge kleine Dinge zu beachten gibt, die man aber wunderbar mit den Skript-Fehlermeldungen abfangen kann. Zunächst wollen wir ja einen String aus einem festen Text und dem Schaden des Spielers generieren. Daher müssen wir den Text und den Schaden per +-Operator zusammenfügen. Da der Schaden aber vom Typ NUMBER ist, brauchen wir den Befehl STR, der den Schaden vom Typ NUMBER in STRING umwandelt. Allerdings erwartet STR keinen Code, sondern höchstens eine Variable, kann also selbst keinen Code auswerten, daher mussten wir den Umweg über die Hilfsvariable `_schaden` gehen. Einer der Gründe, warum ihr euch schnellstmöglich den Befehl FORMAT aneignen solltet, auf den wir aber in Teil III – SQF in der Praxis erst ausführlich eingehen. Als letztes muss man noch beachten, dass auch HINT als Argument nur einen fertigen String erwartet. Auch hier führen wir aber erst einen Code aus, denn die Verbindung zweier Strings mit + ist eine Operation und kein fertiges Argument. Hier reicht es aber, das Ganze in runde Klammern zu packen: Zuerst wird in der Klammer ein einziger STRING gebildet und dieser dann dem Befehl HINT übergeben. Alternativ hätte man auch hier zunächst eine Hilfsvariable `_text` anlegen können.

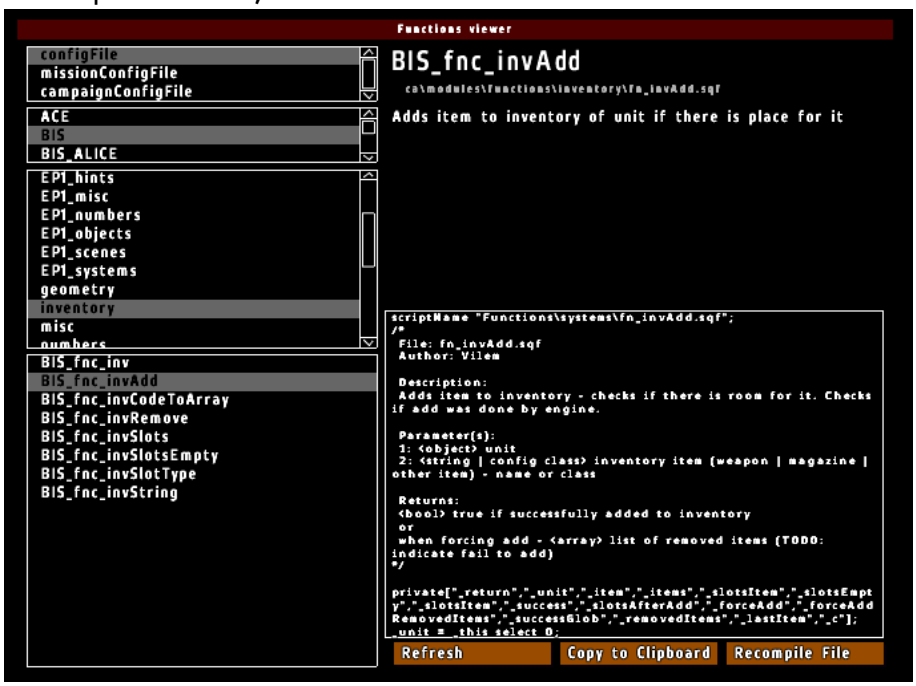
[7] Diese Aufgabe ist nun wieder relativ einfach. Bisher wird der Code nur einmal ausgeführt. Ist eine Einheit zum Zeitpunkt des Auslösen 16 Meter vom Fahrzeug entfernt, passiert nichts, auch wenn sie sich dann weiter nähert. Das wollen wir natürlich nicht! Daher packen wir den oberen Code-Teil, der von Zeile 2 bis 7 geht, in eine while-Schleife, die solange läuft, wie das Fahrzeug eben nicht beschädigt ist:

Loesung_4.sqf
```
1  _grenzabstand = 15;
2  while {damage auto == 0} do
3  {
4  … //Code von Zeile 2-7 aus vorheriger Übung
5  };
6  … //Code von Zeile 8-15 aus vorheriger Übung
```

Streng genommen ist der Code jetzt redundant, da wir den Schaden bereits in der IF-Abfrage mit abfragen. Auf diese und weitere Feinheiten wollen wir aber an dieser Stelle nicht weiter eingehen.

Kapitel I. 6

[1] Wir folgen dem Hinweis und gehen davon aus, dass wir die Funktion ebenfalls im Umfang des Grundspiels finden, also bleiben wir bei BIS und wählen als Kategorie *inventory*.



Das sieht schon Mal sehr vielversprechend aus! Wir haben eine Funktion gefunden, die ein Item in das Inventar der Einheit hinzufügt und sogar prüft, ob dafür Platz ist. Wir lernen aus Feld 6 außerdem, dass es zwei Parameter für den Aufruf der Funktion gibt: 1. die Einheit und 2. den Klassennamen des Items als String. Als Hinweis waren beide Klassennamen bereits vorgegeben und somit lautet die Lösung der Aufgabe:

```
_status = [player, "Binocular"] call BIS_fnc_invAdd;
_status = [player, "ACE_M136_CSRS"] call BIS_fnc_invAdd;
```

Wir speichern den Rückgabewert in der lokalen Variablen `_status`, damit wir überprüfen können, ob das Hinzufügen der Ausrüstung wirklich funktioniert hat. Die Überprüfung sparen wir uns hier.

[2] Jetzt seid ihr also dran! Wir haben mit euch die Funktion bis hierhin entwickelt. Zunächst müssen wir uns überlegen, was an der Funktion überhaupt angepasst werden muss. Da zum Glück hier nur nach der Entfernung des Fahrzeugs zum Spieler gefragt ist, brauchen wir keinen neuen Übergabeparameter, denn PLAYER ist (im SP) immer eindeutig und als Befehl verfügbar. Ansonsten hätten wir auch die Distanz zu einem beliebigen Objekt bestimmen können. Das wäre dann aber als neuer Übergabeparameter mit hinzugekommen und wir hätten noch mehr an der Funktion anpassen müssen.
Daher können wir direkt zum Rückgabewert gehen. Dieser ist momentan definiert durch `_aktuelleAnzahl;`. Daher müssen wir an dieser Stelle unsere gewünschte Entfernung hinzufügen. Die Entfernung als solche bestimmen wir ganz leicht mit `player distance _vehicle`. Da wir zwei Rückgabewerte haben wollen, reicht aber die Dimension nicht mehr aus, wir brauchen ein ARRAY. Das war aber auch das ganze Geheimnis:

```
[_aktuelleAnzahl, player distance _vehicle]; // 2 Rückgabewerte
```

Sind wir damit schon ganz fertig? Nur was die Funktion angeht! Funktioniert jetzt aber noch unser Hauptskript „freieFahrzeuge.sqf“? Leider nicht, denn wir haben ja den Rückgabewert verändert! Unsere gesuchte Anzahl an freien Sitzplätzen war bisher der einzige rückgabewert, damit konnten

wir ihn direkt verarbeiten. Jetzt ist er das erste Element eines Arrays, wir müssen also folgende Zeile im Hauptskript ebenfalls anpassen:

```
if (_x call fnc_freiePlaetze == 0) then // alte Zeile
if (((_x call fnc_freiePlaetze) select 0) == 0) then // neue Zeile
```

Lästige Klammerarbeit. Einige Klammern können mitunter weggelassen werden, da der Compiler bereits eine bestimmte Reihenfolge von Befehlen einhält, wir raten aber explizit zum logischen Klammern von zusammengehörigen Ausdrücken.
Jetzt haben wir die Distanz zwar noch nicht ausgegeben, das könnt ihr aber selber einmal probieren.
Kommen wir zum zweiten Teil der Aufgabe. Gefragt war nämlich noch nach einer Lösung für das Ausklammern von Feindfahrzeugen, da diese ja doch etwas unerwünscht sind, wenn es um freie Fahrzeuge geht (wie man es nimmt…). Dies ist sicherlich etwas, dass wir **nicht** in der Funktion anpassen oder ändern müssen, denn diese hat mit dem Übergabeparameter nichts zu tun. Wir müssen uns vorher Gedanken darüber machen, welche Fahrzeuge überhaupt an die Funktion übergeben werden sollen. Bisher waren dies alle Fahrzeuge, die VEHICLES liefert, bereinigt um die Benzinkanister. Jetzt wollen wir – als eine mögliche Lösung – unsere FOREACH-Schleife so erweitern, dass vor der Übergabe an die Funktion zunächst geprüft wird, ob das Fahrzeug feindlich ist:

Loesung_5a.sqf

```
1  {
2    // Wenn Fahrzeugplätze = 0 -> Löschen aus Liste
3    if ( ((_x call fnc_freiePlaetze) select 0) == 0) then
4    {
5      _fahrzeugListe = _fahrzeugListe - [_x]
6    }
7  } forEach _fahrzeugListe // alter Code
```

Loesung_5b.sqf

```
1  {
2    // Wenn Fahrzeug feindlich -> Lösche aus Liste
3    // Wenn Fahrzeugplätze = 0 -> Löschen aus Liste
4    if ((side _x == EAST) || ((_x call fnc_freiePlaetze)
     select 0) == 0) then
5    {
6      _fahrzeugListe = _fahrzeugListe - [_x]
7    }
8  } forEach _fahrzeugListe
```

Dies ist die FOREACH-Schleife aus dem Hauptskript. Die einzige Änderung erfolgte in der Bedingung in Zeile 3. Vorher wurde direkt nur der Rückgabewert auf 0 überprüft. Jetzt wollen wir auch Fahrzeuge ausschließen, die auf der OPFOR-Seite sind. Dazu brauchen wir nur die Bedingung zu einer oder-Abfrage (||) erweitern. Das Ergebnis ist jetzt ein Array, das nur noch leere oder freundliche Fahrzeuge enthält.

Kapitel I. 8

[1] Wir hatten bereits gesehen, dass der EH „Fired“ unter anderem den Klassennamen der abgefeuerten Waffe mitliefert. Daher müssen wir „nur“ noch anhand dieses STRINGs entscheiden, ob die Waffe schallgedämpft ist. Das ist immer dann der Fall, wenn die Waffe das Kürzel SD enthält. Jetzt können wir mit dem Funktions-Viewer eine Funktion suchen, die es ermöglicht, in Strings zu suchen. Diesmal hilft uns BIS leider nicht weiter, denn hier finden wir unter der Kategorie Strings nichts Brauchbares. Erst das Addon CBA bietet unter Strings die Funktion CBA_fnc_find. Kopiert euch die Funktion in die Zwischenablage und schaut sie euch im Texteditor an. Es sind ausführliche

Aufrufbeispiele dabei. Damit könnte unser Skript dann so aussehen (wir kopieren hier nicht mehr den Kopf, das haben wir bereits in Kapitel 0 getan):

Loesung_6.sqf

```
_treffer = [_weapon, "SD"] call CBA_fnc_find;
// Auswertung der Suchergebnisse
if (_treffer == -1) then
{ hint "Waffe ist keine SD-Waffe" }
else
{ hint "Waffe ist schallgedämpft!" }
```

Die einzige Schwierigkeit hier bestand darin, die richtige Funktion für unsere Aufgabe zu finden und die Syntax korrekt anzuwenden. Der Rest ist Standard-SQF und bedarf keiner weiteren Erklärung.

[2] Wir wissen selbst nicht, wieso wir so umfangreiche Übungen stellen…fangen wir an! Der gesuchte EH ist „Engine“. Dieser EH ist erfreulich einfach und bietet nur zwei Parameter: `[vehicle, engineState]`. Damit ist klar: der EH muss einem Fahrzeug zugewiesen werden. Wir können jedes beliebige Fahrzeug wählen und weisen den EH dem Fahrzeug in der Init-Zeile zu. Der EH feuert sowohl beim Anschalten als auch beim Abschalten des Motors:

```
number = this addEventHandler ["Engine", {(_this) execVM "navi.sqf"}];
```

Damit ist alle Vorarbeit geleistet, den Rest muss das Skript `navi.sqf` erbringen. Wir präsentieren euch hier eine mögliche Lösungsvariante – mit einem Augenzwinkern  .

Loesung_7.sqf

```
// Passed array: [vehicle, engineState]
_vehicle = _this select 0;
_engine = _this select 1;

if (_engine) then
{
  while {speed _vehicle > 0} do
  {
    fahrtzeit = time; // muss global sein!
    _v = speed _vehicle;
    _tank = fuel _vehicle;
    _huelle = damage _vehicle;

    _meldung = ("Sie fahren momentan " + str(_v) + " km/h \n" + "Der
    Tank ist zu " + str(_tank*100) + " % gefüllt \n" + "Die Hülle des
    Fahrzeugs ist zu " + str((1-_huelle)*100) + " % intakt.")
    hint _meldung;
    sleep 2 // Wir wollen kein Navi, dass uns 30 mal pro Sekunde
    informiert…
  }
}
else
{
  // Nach Abstellen des Motors
  hint (  "Danke für die Fahr mit dem Navi James3000! \n" + "Ihre
  Fahrtdauer betrug "+str(time-fahrtzeit)+" Sekunden\n" + "Ihr Ziel ist
  noch "+str(player distance ziel)+" m entfernt.")
}
```

Wir haben hier zwei neue Befehle benutzt: TIME und FUEL. TIME zählt die Zeit seit Missionsbeginn und ist hier völlig zweckdienlich. FUEL ist wie DAMAGE und SPEED ein einfacher Befehl, der die Tankfüllung als NUMBER zurückgibt. Die Meldung selbst haben wir zur besseren Lesbarkeit in einzelne Strings aufgeteilt

A. 2. VERWENDETE SCRIPTBEFEHLE

A. 3. DATENTYPEN

A. 4. STICHWORTVERZEICHNIS