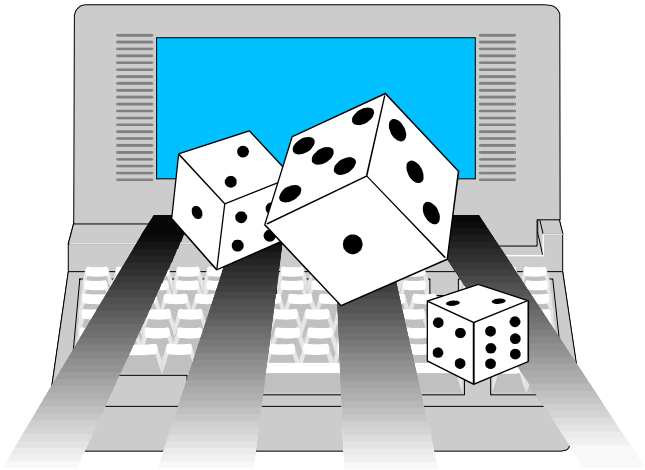
Dice! Plus



Script Tutorial

Contents

Getting Started With Scripts

Scripts are, in their simplest form, just lists of commands that tell *Dice! Plus* what to do. Since you already know how to write down a die roll, you are half way to the simplest form of a script.

This short tutorial will give you an introduction to the possibilities available with the *Dice! Plus* script language.

In this manual, I use several particular fonts and text styles to emphasize certain points. Generally, a monospaced font is used in the example scripts, so it would look like this:

```
RESULT = “sample script”
```

Within the sample script text, a word that is in all capital letters, such as the word RESULT above, means that the word is a command or key word in the script language.

The Simplest Useful Script

If your game book refers to 3d6, you know that it means roll three six-sided dice and add them together, using the result of that addition for the task you are trying to accomplish. Using a simple script is the same thing, except that you have to tell *Dice! Plus* (DP) what to do with the 3d6 that it will be rolling. Here is the simplest possible script, that actually does something useful:

```
RESULT = 3d6
```

This script does just what you would do: it rolls three six-sided dice, adds them together, and shows you the result so that you can do something with it.

Let’s take a closer look at the structure of this one line of script, since it is the basic structure that will be used to do most of the work in the DP script language.

We have three pieces in this script, the ‘RESULT’ piece, the ‘=’ piece, and the ‘3d6’ piece. The ‘3d6’ piece just tells DP to roll 3d6. The ‘=’ piece tells DP that you want to take the piece to the right of it, and

stuff it into the piece on the left of it, which is the ‘result’ piece. The ‘RESULT’ piece tells DP that you want to see the result in the Result box of *Dice! Plus*.

In other words, our tiny, one line script just tells *Dice! Plus* to roll 3d6 and show you the result. Couldn’t you just enter 3d6 into the Die Value / Script box and accomplish the same thing? Yes, you could, but the power of scripting is that you can get much more complicated than the tiny script we are starting with. The Die Value / Script box can only hold one line to roll, but a script can be as long as you need to get the result that you want.

Power Through Little Boxes?

Yup, we’re going to get extra power for our scripts by using little boxes to hold things until we need them. They aren’t really little boxes, but they work pretty much the same way as boxes do; that is, they hold things until you need them, and you can write names on them so that you remember what’s supposed to be inside.

The little boxes I’m talking about are called *variables*. A variable is just a fancy word for a feature that works like a box: it holds something, but what it holds can change if you put something else into it.

In *Dice! Plus*, you can use variables to hold any numbers that you want, even the results of a die roll or a function. However, before you can use a variable, you have to tell DP that you need one, and what it is going to be called. This is just like making a box and writing on it what’s inside before you start packing it full of stuff. Here is how you make your handy box in DP:

```
VAR .variable.
```

So what does this line mean? The ‘VAR’ piece of the line tells DP that you want to make a variable, and the ‘.variable.’ piece of the line tells DP what the name of the variable is going to be. Just like your box, you make it and you name it.

NOTE: Variables can have any name that you want to give them, but it is usually best to begin and end all of your variable names with a common character, such as the periods beginning and ending the name ‘.variable.’ in the example above. Why should you do that? Because if

you don't, DP can sometimes become confused when one variable name includes the name of another variable, such as a variable name of 'dieroll' including the variable name of 'die'. It takes very little effort to enclose all of your variable names in periods or parenthesis, so you should try to do that to protect your variables. Think of it as bubble wrap or foam peanuts inside your boxes to protect the valuable items you have placed inside.

Now that we know how to make our boxes and name them, how do we cram our variables full of neat stuff? Well, believe it or not, you already know how to do it, because our very first sample script showed us the way.

To refresh our memories, here is our original sample script:

```
RESULT = 3d6
```

Remember that the '=' piece of the script line tells DP to cram what's on the right side into what's on the left side. In this sample, we saw that cramming '3d6' into RESULT meant that the result of rolling 3d6 was shown in the Result box. Well, what if we had a handy little variable sitting on the left side instead of the 'RESULT' piece? Wouldn't it get filled up with the result of the '3d6' piece instead? It sure would.

Let's take a look at a sample script that combines our two previous examples, with the minor change of using our variable to hold our 3d6 result:

```
VAR .variable.
.variable. = 3d6
RESULT = .variable.
```

Well, our script has gotten longer, even if it still doesn't do much. We know that the first line makes our variable, and names it. Our second line rolls 3d6 and puts the result into our variable for safe keeping. Our third line we haven't seen before, but it isn't doing anything new; it just takes what we have stored away in our little box and displays it in the Result box.

It is important to remember that when you use a variable in a script, DP is really looking into your little box and using what it sees inside. This means that in our sample script above, when DP rolls the 3d6, it gets some result, let's say 11 for now, and stores that number in the

variable. Now, our variable is holding the number 11, ready for us to use it somewhere else. Then, in the last line of the script, when we tell DP to take the variable and display the result in the Result box, DP looks into our variable, sees the 11 sitting there, and hands it over. Don't worry about RESULT taking away the 11 from '.variable.' though, because it is still inside. Our boxes are a little magical, so they can give out what they are holding every time you want one. No matter how often you get a number from a variable, it always has that number to give out again. The only time our boxes will lose what they are holding inside is if you tell them to hold something else instead.

Garbage Chutes

By now, you have probably figured out that the 'RESULT' piece we used in our sample scripts is a kind of variable. You're right, it is. RESULT is a special kind of variable, called a *result variable*, that tells DP that you want to display what it is holding in the Result box. However, RESULT isn't a very good box, because you can't take out what you put into it, you can only put more stuff in. RESULT is actually more like a garbage chute: you can dump stuff in, but you can't get it out again.

We also have a second garbage chute in the DP script language, that works a lot like RESULT does. In fact, this result variable is a big brother to RESULT called WINRESULT. What WINRESULT does is the same as RESULT with one additional feature: it causes a little message box to pop up on the screen containing what you dumped down the WINRESULT chute.

Our two result variables do have one other feature that makes them better than our regular variables, though. You can shove text instead of just numbers into them. This means that you could have a script that gives you a short message instead of just showing you a number. That could be pretty handy, don't you think?

To make RESULT or WINRESULT display a short message instead of a number, just put the text inside of quotes after the '=' piece of the script line. For example:

```
WINRESULT = "This is a short message"
```

will cause a message window to pop up on screen, displaying the message “This is a short message” to the user.

There are a couple of other things you need to know about our special garbage chute variables. The first is that only the last one you used will cause anything to happen. In other words, if you used this script:

```
WINRESULT = “This is a short message”
RESULT = 5
```

the only thing that will happen is that a ‘5’ will appear in the Result box. The message window will not appear because only the last result variable that got stuff dumped into it will cause anything to happen.

The second thing you need to know is that they don’t actually cause anything to happen at all. Anything you dump into one of the chutes just sits there until the script ends. Once the script ends, and something was dumped into a chute, the proper action occurs, depending on which chute was used last.

Garbage On Display

The fact that you only see what was dumped into a result variable when the script ends wouldn’t be that big a deal in a simple script, but in the more complicated scripts we’ll get to later, that could be a big problem, because a script doesn’t have to end! So we have a special script command that tells *Dice! Plus* that it should show the world what you dumped down the garbage chute. That command is DISPLAY.

DISPLAY sits by itself in a script line, because it doesn’t do anything to anything else. DISPLAY just tells DP that it needs to show you the results you have stuffed into your result variable of choice. For example, if we use the sample script from a few paragraphs back, and add a DISPLAY to it, we can see the message that we wanted to display in the WINRESULT window. For example:

```
WINRESULT = “This is a short message”
DISPLAY
RESULT = 5
```

would result in a window popping up on screen containing the message “This is a short message” and waiting for you to click Okay to make it go away. Then, the Result box would get updated with the number 5 because you dropped that into the RESULT chute just before the script ends.

Now, if the RESULT and WINRESULT lines in the example script above were switched, you would never get to see the ‘5’ in the Result box, because it would be instantly replaced with the message from the WINRESULT result variable. This is because the window that pops up to show you what is in WINRESULT will wait for you to make it go away, allowing you to see the result. On the other hand, RESULT just updates the Result box, which happens instantly, and is quickly replaced by the value that would be displayed by WINRESULT when the script ends.

If We Only Had A Brain

So far, what we have learned to do is pretty straight-forward stuff, that doesn’t necessarily add much to the power of *Dice! Plus*. That’s about to change, though, because DP does indeed have a brain.

To make scripts that are capable of doing neat things, you need to have the ability to make decisions. How can you make decisions, if you can’t ask any questions? You can’t. *Dice! Plus* allows you to ask questions and make decisions through the use of the IF..ENDIF block. Basically, this allows you to ask DP true or false questions.

The first part of the IF..ENDIF block is a line starting with the word ‘IF’, which tells DP to look at the rest of the line and answer the question. The question has to be one that DP can answer as True or False, so that you and DP will know what to do next. Normally, these questions are ones that compare one number or variable to another, such as:

- using the ‘=’ sign to compare if the two items are equal to each other, and if they are, DP answers True, otherwise it answers False.
- using the ‘<‘ sign to compare if the item on the left is less than the item on the right, and if it is, DP answers True, otherwise it answers False.
- using the ‘>’ sign to compare if the item on the left is greater

than the item on the right, and if it is, DP answers True, otherwise it answers False.

- using the '<=' sign to compare if the item on the left is less than or equal to the item on the right, and if it is, DP answers True, otherwise it answers False.
- using the '>=' sign to compare if the item on the left is greater than or equal to the item on the right, and if it is, DP answers True, otherwise it answers False.
- using the '<>' sign to compare if the items are not equal to each other, and if they aren't, DP answers True, otherwise it answers False.

Then, if the answer to the question was True, *Dice! Plus* does all the lines of the script contained in the IF..ENDIF statement block, otherwise it jumps to the end of the block, where the ENDIF line is, and continues the script from there.

So what is an IF..ENDIF statement block? Well, the structure of the IF..ENDIF block is such that the IF part (including the question part) is on a line by itself, and the ENDIF part is on a line by itself as well, like this:

```
IF <question part>
   <statement block>
ENDIF
```

So, the IF..ENDIF block is made up of every line starting with the IF part and ending with the ENDIF part. All of the lines that are between the IF part and the ENDIF part make up the statement block. These lines are *only* looked at if the answer to the question part of the IF statement is True.

So, if the question part of the IF statement is True, the lines between the IF part and the ENDIF part are used; otherwise *Dice! Plus* just jumps down to the ENDIF part, and continues on from that point in the script. This is why the IF..ENDIF statement is written with those two little dots between the two parts: to remind you that it is a block with statements between the two parts of the command.

Let's take a look at what an IF..ENDIF block would look like in a short sample script:

```
VAR .variable.
.variable. = 3d6
IF .variable. < 6
    RESULT = “The result is under 6”
    END
ENDIF
RESULT = “The result is 6 or more”
```

As you can see, we now have a script that is capable of doing something more complicated than we could get just typing things into the Die Value / Script box. This script can make a decision and give us a different result based on that decision.

So what is going on in this script? Well, the first two lines you know from our samples earlier.

The third line is the one we are really interested in, because it is the start of our IF..ENDIF block. This line looks inside the .variable. and sees if the number inside is less than the number 6. If the value of .variable. is less than six, the answer to the comparison question is True, so the lines in the statement block are executed. If the answer to the comparison question is False, the script would continue after the ENDIF part of the IF..ENDIF block.

So, lets pretend that we run this script, and *Dice! Plus* rolls the ‘3d6’ part to get a result of 12. Now the 12 is stuck into the variable named .variable., and DP reaches the IF statement. DP looks at the comparison question of “12 is less than 6” and answers False, just like you or I would. Now, because the answer was false, DP skips all the lines until it reaches the ENDIF part, where it once again pays attention to the script lines. Now, it sees the line telling it to dump “The result is 6 or more” into the Result box, so it does it, and then the script ends.

Now, lets pretend that we run this script again, and this time DP rolls the ‘3d6’ part to get a result of 5. Now the 5 goes into the variable, and the comparison question for the IF statement of “5 is less than 6” is answered True. Because the answer was True, DP looks at the next lines of the script, inside the statement block, which it skipped last time. This time, it sees the line telling it to dump “The result is under 6” into the Result box.

Then, it does the next line, which is just ‘END’. What does this line mean? Well, this line tells *Dice! Plus* to stop the script right on that line. Doing this will cause the Result box to be updated, because the script is

over, and it will prevent DP from continuing the script beyond that point.

Stopping this script inside the statement block is important, because otherwise DP would finish the script lines inside the statement block, and then continue on with the lines outside the statement block, which we don't want to happen because that would give us an incorrect result. It is very important that the IF..ENDIF block only tells DP to execute the lines inside the block if the answer to the question is True, it *does not* tell DP anything about any other part of the script. This means that any lines that aren't inside a statement block will be executed by *Dice! Plus*.

I can't stress this hard enough: **any lines that you only want *Dice! Plus* to run when a certain condition is True must be inside a statement block!** Otherwise, DP does not realize what you are trying to do, and happily runs through the lines. Also, any lines inside a False statement block are *invisible* as far as DP is concerned, so even if you have an END command inside a statement block, DP will ignore it unless the statement block is currently True.

It is possible to nest one or more IF..ENDIF blocks inside of another, and this can be a very useful tool. For example:

```
VAR .variable.
VAR .random.
.variable. = 3d6
IF .variable. < 11
    .random. = 1d10
    RESULT = “You get an empty box”
    IF .random. < 8
        RESULT = “You get a strange potion”
    ENDIF
    IF .random. < 5
        RESULT = “You get a +1 magic sword”
    ENDIF
    IF .random. = 1
        RESULT = “You get a +2 magic sword”
    ENDIF
    END
ENDIF
RESULT = “Your task failed”
```

This is a somewhat more complicated script, but only because there is more to it. Everything in this script, you already know how to do; the only difference is that you have some IF..ENDIF blocks nested inside of

another one.

In this script, if variable is 10 or less, the lines inside of the first block will be executed, otherwise only the last line will generate a result.

Also, notice how the IF..ENDIF blocks inside the first block are arranged. They are arranged from larger result to smaller result. Why? Because this will help to make sure that only the correct result ends up in our Result box. Let's say .random. held a value of 4. The first line after assigning the 4 to the .random. variable is a simple assignment to the Result result variable. Next, we see that 4 is less than 8, so we get a new assignment to the Result box. Then, we see that 4 is less than 5, so we get yet another Result assignment. The last IF is false, so that one is skipped. Then, the 'end' terminates the script, and we are left with the result that we wanted, which came from the 'if .random. < 5' line.

Now, if those IF..ENDIF blocks had been in reverse order, with the smaller item comparisons at the top, what would have happened? Well, any result less than 8 would have been assigned the "You get a strange potion" result when we clearly wanted the lower numbers to get different results! This is not our intention, so we arranged the items from larger to smaller.

Please note: The arrangement from larger to smaller is necessary because this script was designed to 'fall through' all applicable items. This is because we may want to use lines lower down, such as the 'end' line, to apply to all previous lines. If you don't do this, and you would rather have each separate block use it's own END command, then you should be sure to pay attention to the logic of the script flow. In this case, if each block had its own END, then the order of IF questions should be from the lowest to the highest, just the reverse of what it is now.

Improving The Brain

Now that we have seen how we can use IF..ENDIF blocks to make decisions, we have another powerful tool for making our decisions a little easier to understand: the SELECT CASE block. The SELECT CASE block is similar to a bunch of IF..ENDIF blocks that all make comparisons with the same value, except that it is structured a little bit differently and allows us to use a somewhat easier to read and under-

stand structure. Let's take a look at the sample script we used in the IF..ENDIF example above, except we'll rewrite it to use a SELECT CASE block instead. Here it is:

```
VAR .variable.
.variable. = 3d6
IF .variable. < 11
   SELECT CASE 1d10
      CASE ELSE
         RESULT = "You get an empty box"
      CASE IS < 8
         RESULT = "You get a strange potion"
      CASE IS < 5
         RESULT = "You get a +1 magic sword"
      CASE 1
         RESULT = "You get a +2 magic sword"
   END SELECT
   END
ENDIF
RESULT = "Your task failed"
```

The first thing you should notice is that we no longer have the 'VAR .random.' line, and we don't use the variable .random. anywhere in our script. Before, we needed something to hold the result of our 1d10 roll so we could make sure that all our IF statements were looking at the same number. Using SELECT CASE, all of our comparisons are made to the number used in the SELECT CASE line.

So, what is going on in our new script. First, the 'SELECT CASE 1d10' line rolls a 1d10, and remembers the result so that it can be used in each of the following comparisons. Then, we have a bunch of CASE <something> lines, followed by an END SELECT line. The END SELECT line just tells DP that the SELECT CASE block is over, so it can go back to running the script without looking for CASE lines.

In between the SELECT CASE lines and the END SELECT lines is where all the work is being done, and it's the CASE lines that are doing the decision making.

Let's start at the bottom, with the 'CASE 1' line. What this line does is look at the number stored away in the SELECT CASE line, which in this case is the result of a 1d10 die roll, and see if it is equal to 1. That's it. CASE followed by a single number just means "see if the number we're comparing is equal to the number on this line." If it is, then DP

will execute the lines in the block after the CASE line, until it reaches either another CASE line, or the END SELECT line. If the number we're comparing isn't equal to the number on the CASE line, then DP will skip all the lines it sees until it reaches another CASE line, or the END SELECT line. Basically, this line is similar to an IF..ENDIF block that looks like this:

```
IF 1d10 = 1
   RESULT = "You get a +2 magic sword"
ENDIF
```

The big difference is that the 1d10 was already rolled, and you don't have to waste time with a whole bunch of IFs and ENDIFs, because a CASE takes care of it for you.

Now, by looking at the middle two CASE blocks in our SELECT CASE block, we can see that we are making comparisons using the '<' sign. If you want to make a comparison that isn't a simple 'is it equal or not', then you have to use the word IS between the CASE and the comparison you want to make. Using CASE IS allows you to use all the same comparisons that you can use with an IF statement, except that the left side of the comparison (where the CASE IS part is located on the line) will be taken up by the comparison value. So, our CASE block of:

```
CASE IS < 8
   RESULT = "You get a strange potion"
```

is about equivalent to the IF block of:

```
IF 1d10 < 8
   RESULT = "You get a strange potion"
ENDIF
```

By now, you can probably see that a few CASE statements and their surrounding SELECT CASE and END SELECT statements make scripts a little easier to write, and with a little practice, a lot easier to understand as well.

The top CASE statement is a little bit different from the others, because it has the word 'else' there instead of a number. What does this mean? A CASE ELSE block in a SELECT CASE block allows you to

have a CASE block inside the SELECT CASE block that will always be run. This allows you to have a default result, in case one of the comparisons below it does not satisfy the value that you are comparing. For example, if the 1d10 roll of the script resulted in an 8, 9, or 10, none of the other CASE lines would apply, so the result made in the CASE ELSE block would be used.

As with the IF..ENDIF blocks that we used above, the CASE blocks are arranged from larger result to smaller result, and for the same reason. The CASE statements will always be evaluated so long as they are True, so you want the ones that have smaller chances to be done last. Note that the CASE ELSE block is at the top; this is because the CASE ELSE is *always* True, and therefore would replace the results of any previous block if it was at the bottom of the block of CASE blocks.

The Case For Real Power

We have just seen the usefulness of the SELECT CASE statement block, because it can make scripts easier to read and understand, and requires less work to use for many comparisons. However, the real power of the SELECT CASE statement is the way it makes so many different comparisons so much easier to use than a bunch of IF..ENDIF blocks.

What if we wanted to change our sample script a little bit, so that we could use a small bell curve to see what we get in the middle section? Let's change it to use 3d6 instead of a d10, and have the cool stuff appear on both ends of the scale. Our new script might look like this:

```
VAR .variable.
.variable. = 3d6
IF .variable. < 11
   SELECT CASE 3d6
      CASE ELSE
         RESULT = "You get an empty box"
      CASE 3, 18
         RESULT = "You get a +2 magic sword"
      CASE 4, 5, 16, 17
         RESULT = "You get a +1 magic sword"
      CASE 6 TO 8, 13 TO 15
         RESULT = "You get a strange potion"
   END SELECT
```

```
   END
ENDIF
RESULT = “Your task failed”
```

As before, we have the CASE ELSE statement block to cover any results that we don’t explicitly check against.

With this new script, however, we are learning two new things that we can do in the SELECT CASE statement, with the CASE statements. First, we can combine several CASE statements into a single one, so long as each value is separated by a comma. We ***can not*** combine a CASE IS statement, however, so if we used any CASE IS statements, the would still have to be done individually. Second, we can check to see if our comparison value is within a range by using a TO between the two ends of the range to compare with. Using TO eliminates the need to list a bunch of individual numbers that simple proceed in order from one end of the range to the other.

Trying to compare a value to several other values all at one time using an IF..ENDIF block would be a major pain, at the least, so the SELECT CASE block grants us a significant bit of power.

Jumping About, And Using Line Labels

Sometimes, you may have some code that you want to go to a bunch of times. So far, you don’t really know how to do that. Well, DP has a script command for that, as well: GOTO. GOTO tells DP that you want to go to someplace else. With the GOTO command, however, you have to be able to tell DP where it is you want to go, so DP also allows line labels. Line labels are kind of like addresses for a GOTO command, but you can use them even if you don’t use any GOTOs.

A line label is on a line all by itself, and it starts with the colon ‘:’ character. Normally, a line label is ignored by *Dice! Plus* when it is running a script, so you can label anywhere you want, if you want to. The only time DP cares about your line labels is when it gets an address for a GOTO command, then it jumps to the line containing the line label that matches the address (ignoring the colon that designates the line label), and continues the execution of the script from that point.

Here is a simple script that makes use of a GOTO and a line label to do some constructive work:

```
VAR .total.
VAR .roll.
.total. = 0
:loop
    .roll. = 1d6
    .total. = .total. + .roll.
    IF .roll. < 6
       RESULT = .total.
       END
    ENDIF
GOTO loop
```

What this script does is pretty simple, but we couldn't do it without our line label and GOTO command: it rolls 1d6, adding the result to the total each time it rolls; if the roll is less than 6, the total is displayed and the script ends, otherwise the script jumps back to the ':loop' line label and rolls another 1d6, which is added to the total.

Please note that the ':' which starts the line label of ':loop' is not part of the name of the label, it just tells DP that it is a line label. This is why you don't use the colon as part of the goto address in the GOTO command.

What this sample script is doing is very much like what you would do if you were rolling a d6 for one of the many game systems that use a wild die. Every time you get a six, you roll again and add the result to your total, until you no longer roll a six.

Readable Script Code (A Small Digression)

Look at the way the sample script used in the previous section is set up, with a check for a non-six result to end the script, instead of a check for a six to do the GOTO command. This setup allows the whole looping chunk of code to have a clearly defined beginning, the ':loop' line, and ending, the 'goto loop' line. Now, when the code between the beginning and ending lines of the loop are indented, it is easy to see that all of the indented code is of the same section of the script, the same level of importance.

By using indentation every time I enter a different block of code, such as that used in all the sample IF..ENDIF blocks, I can make it easier

to see what each piece of my code is up to, because I can clearly see where each chunk of the script code begins and ends. This also allows me to see at a glance which chunks of code are dependent on, or subsections of, other parts of code.

You can also use blank lines to separate the different chunks of your code, as these will be obvious to you, but ignored by the program.

Indentation and blank lines, however, are note the only ways to improve the readability and understandability of your script code. Another way is using comments to explain what the pieces of your script are supposed to be doing.

Comments in the *Dice! Plus* script language are marked by the semi-colon character ';'. Everything on a line that follows a semi-colon is ignored by DP; its sole purpose is to allow you to embed comments into your scripts without confusing *Dice! Plus*.

The sample script of the last section, with comments and a blank line or two added, might look something like this:

```
;define all our needed variables
VAR .total.
VAR .roll.

;start the executable section of code
.total. = 0
:loop        ;top of the loop
   .roll. = 1d6
   .total. = .total. + .roll.
   IF .roll. < 6
      ;if the roll wasn't a six, display
      ;the total and end
      RESULT = .total.
      END
   ENDIF
GOTO loop ;bottom of the loop
```

While these additional comments may not be that important for such a small script as this one, more complicated scripts will greatly benefit from the additional help that a few well placed comments will confer.

Remember, anything following a semi-colon on the same line is treated as a comment, so while you can put comments after commands on the same line, you can not put commands after a comment on the same line, because they will be ignored.

Looping A Certain Number Of Times

Now that we know how to use a simple loop, what do we do if we want to run a loop a specific number of times, and then stop? Well, using what we already know, we can make a simple script, like this one, that rolls a die six times, showing the result each time:

```
VAR .counter.
.counter. = 0
:loop
    .counter = .counter. + 1
    IF .counter. > 6
       GOTO afterloop
    ENDIF
    WINRESULT = 1d6
    DISPLAY
GOTO loop
:afterloop
RESULT = .counter.
```

This script simply keeps a counter that adds one to itself every time the loop starts over, and when it reaches 7, instead of rolling the 1d6 and displaying the result, it jumps out of the loop and displays the value of '.counter.' so you know it's done.

This sample script really isn't that useful, of course, but it is a handy way to see what extra steps are necessary to do a limited run loop such as this one. From looking at the sample script, we can see that doing a limited run loop such as this one requires at least 7 lines of code (two line labels, two GOTO commands, the IF..ENDIF lines, and the line that increases the counter variable). If we use very many loops of this kind, that's a lot of extra lines to put into a script. There must be a better way, and there is: the FOR..NEXT loop.

The FOR..NEXT loop is, as you can see from the pair of little dots in the name, another block structure, which means that the FOR line is a the top, the NEXT line is at the bottom, and the statement block will fall into the middle.

What a FOR..NEXT loop does is take away the code that you need to write to tell the script where the loop begins and ends, what to add to your counter variable each time through, and when to stop the loop and continue on with the rest of the script. The way it does this is very

simple:

```
FOR <counter> = <startval> TO <endval> STEP <stepval>
   <statement block>
NEXT
```

The NEXT part of the block just sits there by itself, telling the script that when it reaches this line, it should jump back up to the FOR line. This means that you don't have to include any other lines, or a line label, to get the loop to work.

The <statement block> part, of course, is where you put the code that you want to be executed each time the loop runs. From the example above, this would be nothing more than this pair of lines:

```
WINRESULT = 1d6
DISPLAY
```

since, these lines were all that was done inside the loop that wasn't for the purpose of keeping track of the loop itself.

The FOR line is the complicated part of the FOR..NEXT loop block. The <counter> part of the line is where you would put the counter variable for your loop, this was '.counter.' in the example above. The equals sign in this line tells DP that each piece of the rest of the line will get put into the <counter> part at some point. The <startval> part will be the value that you want the <counter> part to have the first time the loop starts, which would be 1 from our example above. The <endval> part should be the value that you want DP to look at, to see if the loop is finished yet. The <endval> part should be the value that you want to end the script after, so in our example above, this would be replaced with a 6, since we wanted to count from 1 to 6. The STEP part in the line tells DP that we are going to tell it how much to add to the <counter> each time we come back in the loop, and the <stepval> is the amount that will be added to <counter> each time. In our example above, <stepval> would be 1.

Let's take a look at what our example from above would look like, using this new FOR..NEXT loop:

```
VAR .counter.
FOR .counter. = 1 TO 6 STEP 1
```

```
   WINRESULT = 1d6
   DISPLAY
NEXT
RESULT = .counter.
```

For starters, it's quite a bit smaller, because most of the work of setting values and incrementing our counter variable are taken care of by the FOR..NEXT loop for us.

So, what is going on in this little script is this: The variable '.counter.' is created. *Dice! Plus* sees the FOR line, and makes '.counter.' equal to 1, since that is the <startval>. Then the script goes on to the next two lines, which display the window for the result of the die roll. Then the script goes on to the next line, which is the NEXT command, which causes DP to jump back up to the FOR line, and add the <stepval> to the counter. Since the <stepval> is 1, '.counter.' would then hold a value of 2, and since 2 is less than or equal to the <endval> of 6, the script continues to run the statement block. This goes on right through a value for '.counter.' of 6, because a FOR..NEXT loop includes the <endval> value as the last value it will use, so that it will exit only when the counter value exceeds the <endval> value. So, when the counter value is 6, the script runs down to the NEXT line once again, then jumps to the FOR line. At this point, DP adds 1 to the counter again, and then checks to see if it is more than the <endval> of 6. Since the counter is a 7, DP jumps down to the NEXT line, and continues on with the script at the first line after that.

The important thing to remember is that the counter variable is first set to the <startval>, then every time the loop comes up after that, the <stepval> is added on before comparing it to the <endval>. You must remember, also, that the <endval> is the last value that the loop should be run for, not the value that makes the loop end immediately.

Now that you see how to make the FOR..NEXT loop work, it is also easy to make the FOR..NEXT loop even a little bit simpler, since you do not have to include the STEP <stepval> part of the FOR line if you do not want to. If you leave it off, *Dice! Plus* will automatically use a <stepval> of 1, unless the <startval> is higher than the <endval>, in which case DP will use a <stepval> of -1. Yes, you can count backwards in a FOR..NEXT loop. As a matter of fact, this is what our example would look like if we wanted to roll six times, but counting backwards

(and leaving off the <stepval>, which is optional for steps of 1 or -1):

```
VAR .counter.
FOR .counter. = 6 TO 1
   WINRESULT = 1d6
   DISPLAY
NEXT
RESULT = .counter.
```

That looks pretty straightforward, doesn't it?

Of course, you can count by values other than 1, but in that case you must use the STEP <stepval> part of the FOR line.

Let's try something a little more complicated with the FOR..NEXT loop now. What if we want to roll three times, rolling 2 dice the first time, 4 dice the second time, and 6 dice the final time. Let's just adapt our handy example:

```
VAR .counter.
FOR .counter. = 2 TO 6 STEP 2
   WINRESULT = .counter.d6
   DISPLAY
NEXT
RESULT = .counter.
```

Notice that we are now starting the counter at 2, since we want the first roll to be of 2 dice. We set the <stepval> to 2 so that we would add 2 to the counter each time, making it 2 the first time, 4 the second time, 6 the third time, and when it comes back around, the 8 would make the script jump down to the line after the NEXT line.

(One other thing to keep in mind is that the value of your counter variable is actually changed before it is compared to the <endval>, so it will be whatever value it was to make it exit out of our looping (8 in the last example). You can see this if you run this example script in *Dice! Plus*, because after the script is finished, the Result box will hold the final value of '.counter.'.)

Note also that we are using the counter variable as the number of dice we want to roll. This is perfectly acceptable, since '.counter.' is a perfectly normal variable in every way. However, if you change the value of the counter variable inside the loop, you will quite possibly destroy the count that you are trying to maintain, so don't *change* the

value, although you can *use* the value. For example, this script will stay in the loop forever, because it will never get beyond a value of 4.

```
VAR .counter.
FOR .counter. = 2 TO 6 STEP 2
   WINRESULT = .counter.d6
   DISPLAY
   .counter. = 2;bad idea to change the counter
NEXT
RESULT = .counter.
```

In this example, the first run sets '.counter.' to a value of 2, then runs the statement block. Inside the statement block, '.counter.' is set to a value of 2. Then the NEXT makes the jump back to the FOR line, where 2 is added to the value of the counter, which is 2, making a value of 4, which means the loop runs again, where the counter is set to 2, and the NEXT jumps back to the FOR, where 2 is added to the value of the counter, and so on, forever. Not a good thing, and not very useful, really.

Getting Input

While the script language as you've learned it is pretty powerful already, there is another command that you can use to make your scripts a little easier to interact with, and that command is GET. GET allows you to get input from whoever is using your script, so that they don't have to mess with the script itself to do what they want. GET is very easy to use, all you need is a variable. Here is a tiny script that gets input from the user:

```
VAR .user.
GET .user., Enter a number Mr. User:
RESULT = .user.
```

The GET line will pop up a little window that gets a number from the user. It uses the variable .user. to store the number that the user will enter. The part of the GET line after the .user. variable, separated by the comma, is the prompt that will be displayed in the GET window to let the user know what they are entering the number for, or anything else you want to say in the GET window; don't get too wordy though, since

space is somewhat limited. If you don't want to use your own prompt for the GET window, you don't have to. Here is the same script, except that the prompt that will tell the user to enter a number will be made up by *Dice! Plus* instead of using one that you provided:

```
VAR .user.
GET .user.
RESULT = .user.
```

As you can see, using GET is very simple and straight-forward, and provides an easy way to interact with the user of your script.

Ready To Rip

Believe it or not, you have now been exposed to all of the major features of the *Dice! Plus* scripting language. While there are still a few commands that you haven't seen used, these are generally very simple things that you will understand easily just from their description in the Script and Function Reference.

Now, just to prove that I'm telling the truth, let's take a look at one of the sample scripts that comes with *Dice! Plus*: SHDWRUN.DIE, which is a script for counting successes in the Shadowrun game. Here is the script (edited slightly to capitalize all the commands, to conform with the rest of this manual):

```
;  Script for counting successes in Shadowrun. This
;script requires DICE! PLUS v1.2, which should be
;with this script, or available where you downloaded
;this script.
;  v1.2 is required because earlier versions did not
;support use of >= together, or the SELECT CASE
;block statement.

;Define our variables for the script
VAR .dice.
VAR .target.
VAR .counter.
VAR .total.
VAR .success.
VAR .dieroll.
```

```
; Here is the part of the script that we want to have
; running all the time. If you want to run this script only
; once per Roll, comment out the :loop and goto loop
; lines, but remember that you'll have to reenter the
; dice and target numbers every time if you do that.
:loop
   ; Clear the variables for each roll
   CLEARLOG
   .total. = 0
   .success. = 0

   ; Get the number of dice to roll and the target number
   GET .dice., Enter the number of dice to roll (0 to
exit):
   IF .dice. = 0
      RESULT = "terminated"
      END
   ENDIF
   GET .target., Enter the target number:

   ; Roll each die, checking for success
   FOR .counter. = 1 TO .dice.
      .dieroll. = @WILD(d6)
      .total. = .total. + .dieroll.
      IF .dieroll. >= .target.
         .success. = .success. + 1
      ENDIF
   NEXT

   ; Display the results
   IF .total. = .dice.
      WINRESULT = "Botch"
   ENDIF
   IF .total. > .dice.
      SELECT CASE .success.
         CASE IS < 1
            WINRESULT = "No Successes"
            EXIT SELECT
         CASE 1
            WINRESULT = ".success. Success"
            EXIT SELECT
         CASE IS > 1
            WINRESULT = ".success. Successes"
            EXIT SELECT
      END SELECT
   ENDIF
```

```
    DISPLAY
GOTO loop
```

Wow! That's a long one. Let's look at this script one piece at a time. Remember, we can ignore all the comments after the semi-colons, because DP will. They are mostly there to help you see what's going on when you are looking at the script in *Dice! Plus*.

First, though, if you aren't familiar with Shadowrun, the system uses d6, of which you roll a number equal to your skill level. The object is to roll the dice and try to get a number of successes. You get a success when the number on a d6 is higher than, or equal to, the target number for the task. The target number is set by what you are trying to do, and the higher the target number, the harder the task. Also, if you roll a 6 on your d6, you get to roll again and add the result to your 6, doing this again as long as you keep rolling 6 on the die. If all the dice that you are rolling turn up 1's, you have botched your roll.

So, what this script is trying to do is roll a bunch of d6's, rerolling and adding on those that come up sixes, and comparing each die to the target number assigned by the GM. Also, take note if we got all ones so that we'll know that we botched.

Let's get to it, and take a look at the first chunk of the script.

```
VAR .dice.
VAR .target.
VAR .counter.
VAR .total.
VAR .success.
VAR .dieroll.
```

This first chunk of script code is just setting up our variables, so nothing special here. Here is the next chunk:

```
:loop
    ; Clear the variables for each roll
    CLEARLOG
    .total. = 0
    .success. = 0
```

Here, we are just setting a line label, and clearing a couple of variables. The line label is being used by the GOTO at the bottom of the script to make a big loop. The big loop allows us to keep running the

script without having to always hit the Roll button. As long as the script is running, Dice! Plus will keep track of our variable values for us without resetting them to zero, and that is what we want, since we will often be rolling the same number of dice over and over again.

Clearing those two particular variables is important, because they are keeping track of things that should be reset every time the loop starts, unlike the other variables which we want to keep their values unless they are specifically changed. The .total. variable is keeping track of the total of all die rolls made during this pass of the loop, and the .success. variable is keeping track of the number of successes (rolls over the target number) that we have made during this pass of the loop.

We also sneak a new command in here: CLEARLOG. All this command does is empty out the Long Results box, so that it doesn't over flow while the script is running. CLEARLOG is important because with the loop always running, DP doesn't have any other way to know that the Long Results box should be cleared out. Normally, it is cleared at the start of each script, but this one is designed to keep running, so we have to clear out the Long Results manually.

Here is the next chunk of the script:

```
   ; Get the number of dice to roll and the target number
   GET .dice., Enter the number of dice to roll (0 to
exit):
   IF .dice. = 0
      RESULT = "terminated"
      END
   ENDIF
   GET .target., Enter the target number:
```

This chunk starts out by asking the user to enter a number of dice to roll, and stores that entry in the .dice. variable. The first time this script starts up, .dice. won't have a value yet, but for all the following rolls, .dice. will remain set to what ever was entered by the user, so it will default to that value in the GET window.

The script needs an easy way to exit, so we chose to do it by quitting when the user enters 0 dice to roll. So, the IF statement checks to see if the .dice. variable is set to zero, and if it is, the script ends. If not, it continues on to the next GET statement, where we find out from the user what the assigned target number is.

As with the .dice. variable, the .target. variable is not cleared be-

tween loops, so after the first run through of the loop, the GET window will have a default value equal to the current value of the .target. variable.

And now on to the next piece:

```
; Roll each die, checking for success
FOR .counter. = 1 TO .dice.
   .dieroll. = @WILD(d6)
   .total. = .total. + .dieroll.
   IF .dieroll. >= .target.
      .success. = .success. + 1
   ENDIF
NEXT
```

Just like the comment says, this chunk rolls each die, and checks to see if it is a success. To roll each die, we use the FOR..NEXT loop, because it will handle most of the dirty work for us.

Inside the FOR..NEXT loop, the first line assigns the results of a wild d6 roll to the variable .dieroll. Just a second now, what is that odd looking @WILD thing? That is one of the many functions available in *Dice! Plus*. By using the @WILD function to roll our d6, we don't have to worry about checking to see if the roll is a 6, and then rolling again and adding if it is, because that is what the @WILD function does! Using this function here makes our life quite a bit easier.

Just to be a little more specific, since we are using it, what the @WILD function does is this: it looks at the roll that it is being told to make, which is a d6 in this case, and finds the highest result for that roll, which in our case is a 6. Then, it makes the roll, and if the high result comes up, it rolls again and adds the result to the previous roll, doing this until the high result doesn't come up. This is exactly what we need for a Shadowrun die roller.

The line after the .dieroll. variable is assigned from the function just adds the result of the dieroll to the current total of all rolls. We need the total later to see if we have botched (rolled all ones).

The next line checks to see if our dieroll was a success, by using an IF statement to compare the .dieroll. variable to the .target. variable, which was entered at the earlier GET statement. If the .dieroll. is greater than or equal to the .target., we have a success, so we increase the number of successes we have by adding one to our current number of successes. This is why we cleared the .success. variable earlier in the

loop, otherwise we might have a number in .success. already from an earlier roll, which would erroneously inflate the number of successes we actually have.

Now on to the final chunk of the script, which is a little longer than the others, but doesn't really do any more:

```
   ; Display the results
   IF .total. = .dice.
      WINRESULT = "Botch"
   ENDIF
   IF .total. > .dice.
      SELECT CASE .success.
         CASE IS < 1
            WINRESULT = "No Successes"
            EXIT SELECT
         CASE 1
            WINRESULT = ".success. Success"
            EXIT SELECT
         CASE IS > 1
            WINRESULT = ".success. Successes"
            EXIT SELECT
      END SELECT
   ENDIF
   DISPLAY
GOTO loop
```

What this chunk does is display the results of our Shadowrun roll to the user. First, we check to see if we botched, by seeing if the total of all rolls is equal to the number of rolls that we made. (If every die comes up a one, the total is the same as the number of dice rolled). If it is, we send a result of "Botch" to WINRESULT.

Then, we check to see if the total is higher than the number of dice. If we botched, we'll skip this whole IF..ENDIF chunk, and go right to the DISPLAY command near the end, otherwise we'll enter the IF..ENDIF statement block to see how we did.

What we need to know now is how many successes we got, so we use a SELECT CASE statement block to find out, and to set our WINRESULT depending on the answer. You'll notice that our quoted text being sent to the WINRESULT window has the variable .success. inside. This variable will be replaced with the value of .success. before the WINRESULT window appears. So, if we had one success, the

WINRESULT window would display “1 Success”. This is another good reason to make sure that you use a naming standard for your variables that is unlikely to conflict with normal text (which is why I use a period-variable-period format), otherwise you may occasionally find your text replaced by numbers in the oddest places.

You’ll notice that we use a new command here, in each of our CASE blocks. The new command, EXIT SELECT, tells DP to jump immediately to the END SELECT line without looking at any of the other CASE lines. This speeds up our script, because once it finds a True CASE, it doesn’t have to waste time looking at the others.

Note that we don’t have a CASE ELSE statement block, this is because our CASE statements cover the entire possible range of results. If we had a CASE ELSE, we would put it at the *end* of the other CASE statements, because any one of them that was True would set WINRESULT and jump immediately to the END SELECT line, so we wouldn’t need the CASE ELSE at the top. If we did put the CASE ELSE at the top of the CASE blocks, however, we would *not* put an EXIT SELECT command in it, because the CASE ELSE is meant to be used only to cover for values that aren’t caught by other CASE statements, so it should never automatically exit the SELECT CASE statement block.

Finally, after we have exited the SELECT CASE block and the IF..ENDIF block, we get to the DISPLAY, which will pop up the WINRESULT window with our results. We need the DISPLAY because the script never ends on its own, so that’s the only way to see the results.

Then, we get to the GOTO statement, which sends us back up to do our loop over again.

That Is All

While I haven’t covered every command that the script language supports, I have covered all of the major ones, and those most likely to give you trouble as you write your own scripts. The best way to learn how to write scripts of your own is to sit down and write scripts. Start with little ones that only do little things, and don’t use very many script commands. This way, you will get a feel for how scripts are written, and you will become more comfortable with the commands before you jump into the larger scripts. Starting with small scripts will also help limit the

frustration of having something go wrong, and needing to find what part of your script is causing the problem.

When you want to do something and you don't know what command to use, look back through this manual to see if something looks similar, or try refering to the Function and Script Reference (contained within the text file REF.TXT) to get explanations of the commands and functions that are available. I would encourage you to read through the Function and Script Reference at least one time, just so that you are familiar with the different functions and commands that are available. You may even find that there is a function that already does exactly what you need.

Additionally, there are a variety of sample scripts included with *Dice! Plus* to show you how I have done some things, as well as to demonstrate the use of the functions that come built-in to the program.

Good Luck,

Armin D. Sykes

Index

Symbols

C

D

E

F

G

I

L

N

P

R

S

V

W

Miscellaneous Information

Where To Reach Me

Snail Mail: Armin D. Sykes
Miser Software
4130 SW 117th #401
Beaverton, OR 97005

via Internet: armin@misersoft.com