VISUAL GUIDE to

RX Scripting

for Roulette Xtreme - System Designer 2.0

L J Howell
UX Software – 2009

Ver. 1.0

TABLE OF CONTENTS

INTRODUCTION ii
- What is this book about? iii
- How to use this book iii
- Time to start iv

GETTING ACQUANTIED WITH RX SCRIPTING 1
- What is RX Scripting? 2
- What RX Scripting Can Do 2
- The Piece-together Language 2
  - Action Command 2
  - Condition Command 2
- Putting the Pieces Together 3
- Using the Statement Completion 4
- The Initial System Design 6
  - System 6
  - Method 6
- Block Structure 8
- Format Structure 9
- Using Comments 10
- Storing Data Values 11
  - Data Records 11
  - Data Flags 12
- Logical Comparisons for Condition Statements 13
  - Logical expression comparisons 13
  - Count of total comparisons 14
  - Lost bets comparisons 14
  - Won bets comparisons 15
  - Outcome of roulette layout comparisons 15
  - Found within a list of numbers comparisons 16
  - Last Answer comparisons 16
  - Span between 2 numbers on roulette wheel comparison 17
  - Distance between 2 numbers on the marquee board comparison 17
  - Pattern Match comparisons 18

WORKING WITH CONDITION COMMANDS 19
- Initial Condition 20
  - While or If condition? 20
  - Loop Until condition 21
- Continuation Condition 22
  - Using the Else continuation 24
- Block Condition 25
  - Using the Group Condition 25

WORKING WITH ACTION COMMANDS 27
- Place Action 28
  - Put 28
  - Put 100% 29
- Mathematical Action 30
  - Add 30
  - Add 100% 31
  - Subtract and Subtract 100% 32
  - Multiply 32
  - Multiply 100% 33
  - Divide and Divide 100% 34
- Input Action 37
  - Input Data 37
  - Input Dropdown 38
  - Display 39
  - Ask 40

Assign Action 41
Assign action commands that alter Roulette Xtreme 41
Apply En Prison 41
Apply Le Partage 42
Load Double Wheel 43
Load Single Wheel 44
Load No Zero Wheel 45
Stop Session 46
Assign action commands that copy data 47
Copy 47
Copy Last 48
Copy List 49
Copy Neighbors 50
Duplicate 51
Miscellaneous Assign action commands 52
Track Last 52
Clear 53
Set Max 54
Set List 55
Generate Random Number 56
Assign action command for data flags 57
Set Flag 57
Reset All Flags 58
Maneuver Action 59
Maneuver action commands that alters a list in a data record 59
Move List Up 59
Move List Down 60
Locate Number Right 61
Locate Number Left 62
Locate Distance Between 63
Locate Number Backward 65
Locate Number Forward 67
Call 69
Return 69
Exit 70
WORKING WITH ROULETTE LAYOUTS 71
Inside Layouts 72
Individual Straight-up Numbers 72
Split 73
Horizontal Split 73
Vertical Split 74
Street 75
Line (Double Street) 76
Corner 77
Outside Layouts 78
Dozen 78
Column 79
Even Chance 80
Using Layouts with Commands 81
Action Command 81
Condition Command 82
CREATING ROULETTE SYSTEMS 83
Martingale System 84
Expanding on the Martingale System 86
D'Alembert System 88
Reverse D'Alembert System 90
Labouchere System 92
Inside Number System 96
RX SCRIPTING TEMPLATE 103
Template to use when creating new systems 104
Adding the Template to System Editor 105

INTRODUCTION

Welcome to RX Scripting. Using this easy-to-learn programming language, you'll be able to create many roulette systems. This book has been written as a painless introduction to RX Scripting, so you don't have to be a programmer to write roulette systems.

So put away your pocket protector as you will not need them at any time because being a computer geek is not required.

What is this book about?

This book is for people who want to take advantage of Roulette Xtreme's powerful RX Scripting language and create systems to test and use in real online or land casinos.

We don't assume you know anything about programming or scripting language. We do however assume you know a little about the casino game of roulette and all of its different betting combinations and payouts.

If you already know a little about programming, you should know that this book does not take the same approach to RX Scripting as you might learn from other languages. This book does not delve deeply into the RX Scripting language as you can find all the information about the syntax language with the help documentation included with Roulette Xtreme software. However, where appropriate, I will go into detail on some action and condition commands so you don't have to spend a lot of time reading the help documentation. This book concentrates on showing you how to properly understand and use action and condition commands so you can create useful systems with RX Scripting without a lot of extraneous information.

How to use this book

Throughout this book, there are special techniques to make it easy for you to read the book and understand the scripting language.

In the step-by-step instructions that make up most of this book, there are special type styles to denote the RX Scripting code like this:

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
      Put 1 unit on Black
    End
End
```

In the illustrations accompanying the step-by-step instructions, the highlighted parts of the RX Scripting will be in **red** so you can quickly know what example is in discussion. Also note that the RX Scripting language identifiers will be presented in **bold** and the first letter is Capitalize, (i.e. **Method**, **While**, **Begin**). Other words will be in normal text including any identifiers that require text enclosed in quotation marks. Following is an example of an input action command with text enclosed in quotation marks.

```
System "my system"

Method "main"
Begin
    Input Data "Enter your bankroll" to Bankroll
End
```

Since I am referencing the input action command the entire RX Scripting format is in **red**. All other words not part of the RX Scripting language will be in normal text as well which are considered filler-words to help make the RX Scripting language easier to read. (i.e. the word **to** as shown above) When I am referring to language identifiers, I will underline and **bold** them like the following action command: **Put**. This way it is not confusing when explaining about this type of identifier.

Time to start

RX Scripting is very easy to start by creating a simple system that allows you to place bets on the Roulette table. Then later add more complicated stuff as you need it. You don’t have to learn the whole book’s worth of information before you can create Roulette systems.

Of course, every journey begins with the first step of understanding the basics of RX Scripting, followed by understanding action and condition commands, then finally creating a simple system to a more complex one.

GETTING ACQUANTIED WITH RX SCRIPTING

1

For roulette gamblers, the evolution of creating systems has been a mixed blessing. In the early days, creating systems was simple as writing them down on a piece of paper (or back of a napkin). The only way to test these systems was to try it out on a real casino and risk losing your money.

Now with internet gambling becoming very popular, gamblers are now able to test the written systems in practice mode (offered on most internet gambling sites) before risking any money.

Those types of testing are very tedious and time-consuming often leading to making mistakes with hand written calculations and such.

Now with Roulette Xtreme System Designer software, you can easily create systems and run them through a series of many tests to validate your system to ensure it is a winning system before trying it out on any casino whether land or internet.

In this chapter, you will learn what RX Scripting can do and also some of the basics of the RX Scripting language.

What is RX Scripting?

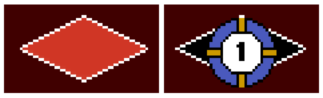
RX Scripting is a programming language that you can use to create roulette systems with Roulette Xtreme System Designer software. But if you are not a programmer, do not panic at the term “programming language”. There are many examples of RX Scripting from different roulette boards especially the ones from the following sites:

- http://www.uxsoftware.com/pages/system.html/
- http://vlsroulette.com/
- http://www.laroulette.it/

The language consists of actions and conditions that allow you to play a created system automatically by placing bets on the table for the development of betting. For each spin the system runs through a series of commands. The structure of the language is much like an English sentence. For example, the code in **red** below:

```
System “my system”
{
 System to place 1 unit on black
}
Method "main"
Begin
   Put 1 unit on Black
End
```

The above RX Scripting code (which is an action command) is telling the system to make a bet of 1 unit to the even chance of black on the roulette table and the result from the code is shown below.



The scripting language is created inside method blocks that have a **Begin** and **End** syntax identifier. The method **main** must exist as this is the entry point where the system starts to run through the series of commands. One method called **Method main** which must exists for every system otherwise the program will fail to compile and work correctly.

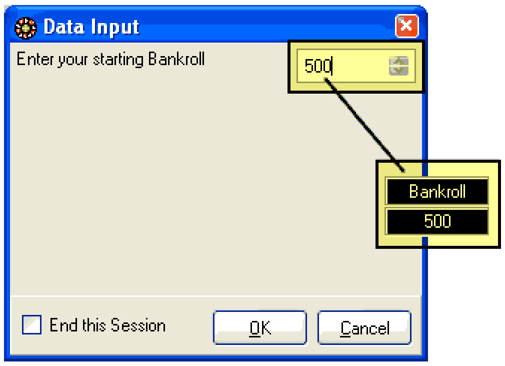
What RX Scripting Can Do

There are many things you can do with RX Scripting when creating a roulette system. RX Scripting lets you create an active user interface, giving you feedback to enter information before or during an active session.

For example, you might want the system to ask you for a starting bankroll (units) at the start of a new session. That's done with the action **Input Data** command. RX Scripting language for this action user interface is shown below.

```
System “my system”

Method “main"
Begin
   While Starting a New Session
   Begin
     Input Data “Enter your starting Bankroll” to Bankroll
   End
End
```

When the script is run, the system presents a user interface dialog screen allowing them to enter a bankroll amount.



You can use RX Scripting to perform complex calculations like tracking the last 37 numbers that have appeared or place bets on different table layouts based on results from certain types of conditions. You can also access statistical data from the outcomes of any of the roulette's layouts such as standard deviation, mean, minimum and maximum and process those statistical data measures to perform decisions of when to bet or when to quit a session.

With RX Scripting, you have the ability to access almost all of the input and statistical functions of Roulette Xtreme software. What that means is you can read through the history of outcomes that have already occurred, access statistical data and perform comparisons of one outcome to another, change the wheel layout from European to American or none and so on.

The Piece-together Language

RX Scripting is an English style language pieced together by different parts to form a complete statement. The first part of a statement is a command which could be an action or a condition type of command. These two types of commands make up the entire RX Scripting language. You perform some action (i.e. placing a bet) or from the results of a condition, perform some action, (i.e. when red has appeared 3 times, bet on black). The following describes the two different types of commands.

Action Command

An action command is an identifier for performing something now without any conditional event to occur. For example: let's say I want to place a $5.00 unit on black. If I was playing a real roulette game, I would put a $5.00 chip on the even chance of black. The process is called an action. I performed such a task without any regard to some conditional event. In this case, the action identifier, **Put** is the command followed by several identifiers pieced-together to form a statement. Below demonstrates how this is done with RX Scripting.

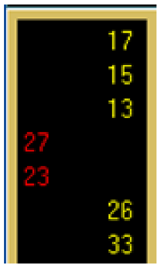
```
System “my system”

Method “main"
Begin
   Put 5 units on the Black
End
```

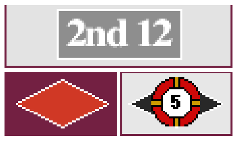
Condition Command

A condition command is a logical process that produces an event of either a true or false. For example: let's say that I want to place a $5.00 unit on the even chance of black only after I noticed that the color black has appeared 3 times in a row. So, I patiently wait spin after spin until I noticed on the marquee board that black has appeared 3 times and once this happens, I immediately place a $5.00 chip on the even chance of black. This is called a condition event where I waited until the event became true before I placed any bets. In this case, the condition identifier, **If** is the command followed by several identifiers pieced-together to form a statement.

The next script example demonstrates how this is done with RX Scripting. The script is using a condition statement to wait for black to appear 3 times in a row before placing 5 units on the even chance of black. As you can see from the marquee board that the last 3 outcomes are black thus causing the condition event to become true which then executes the action statement to automatically place 5 units on the even chance of black as shown on the far right image.

```
System “my system”

Method “main"
Begin
    If Black has Hit for 3 times
    Begin
        Put 5 units on Black
    End
End
```





Putting the Pieces Together

You can put together the different parts to form the English style language to get a better description of what you are trying do.

Let's think about this English phrase.

- **Place 5 chips on the number 21**

Now, everyone can figure out what I mean by the phrase: **Place 5 units on the number 21**. It simply means that I am going to place $5.00 (assuming a unit has the value of $1.00) on the number 21 of the roulette table. So RX Scripting language is very similar in this situation. The difference is the exact wording of the syntax. Here is the exact syntax of the different parts for the above phrase.

```
System "my system"

Method "main"
Begin
    Put 5 Number 21
End
```

Notice that this syntax is a little strange to figure out what it is trying to say. With RX Scripting you can add additional non-functional words to help make the English style sentence make sense. So based on my example of wanting to bet $5.00 on the number 21, I would add some non-functional words like in the next script example.

```
System "my system"

Method "main"
Begin
    Put 5 units on the Number 21
End
```

Now isn't that easier to read and understand? Clearly there are many different ways to express that I want to bet $5.00 on number 21, but for RX Scripting language, there is only 1 way to perform this task. The above language is broken down into 3 parts. The first part is **Put** which is the initial part of an action command. The second part (known as identifier) is **5** which the amount of the bet. The third identifier is **Number 21** which is the roulette number that I am placing a bet on hoping it will appear on the next spin of the wheel. The script below shows an example of making bets on 6 different numbers on the roulette table using the action command **Put** (in this case, to place bets). On every spin, the system places 6 bets on 12,13,19,21,29 and 32.

```
System "my system"

Method "main"
Begin
   Put 1 unit on the Number 12
   Put 1 unit on the Number 13
   Put 1 unit on the Number 21
   Put 1 unit on the Number 29
   Put 1 unit on the Number 29
   Put 1 unit on the Number 32
End
```

15
18
24
27
30
33
11
14
17
20
23
26
10
16
19
22
25
28
31
1
1
1
1
1

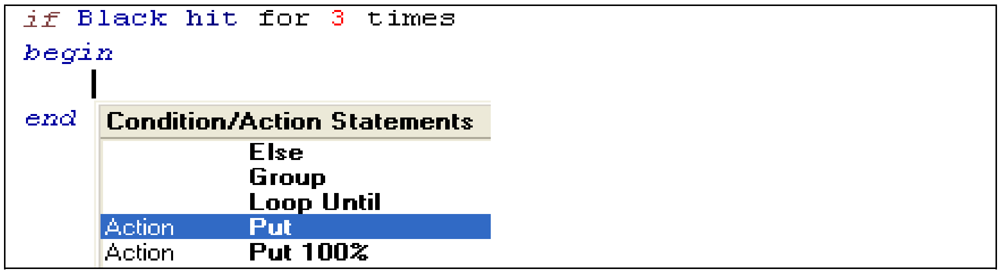
Using the Statement Completion

The built-in statement completion for Roulette Xtreme allows you to create complete action or condition statements that are valid and produce no errors when the system compiles the program. The best way to describe the built-in function is to provide an example:

```
System “my system”

Method “main"
Begin
    If Black has Hit for 3 times
    Begin
        Put 5 units on Black
    End
```

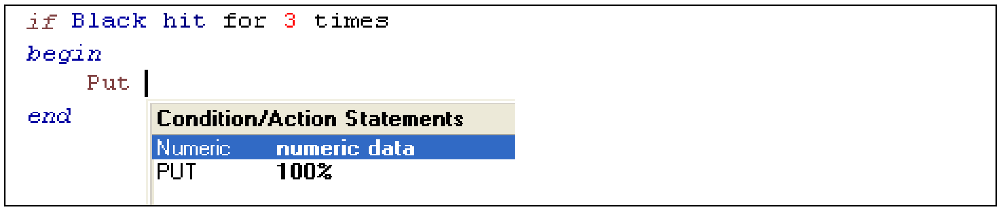
Let’s focus on the action statement: **Put** **5** units on **Black** and create the correct syntax using the built-in statement completion. Assuming that you already entered the condition statement above, perform the following steps:

1. Just below the **Begin** identifier on a new line, press the **F2** key. A valid list of condition and actions commands will be displayed. Find and select the **Put** action, then press the enter key.



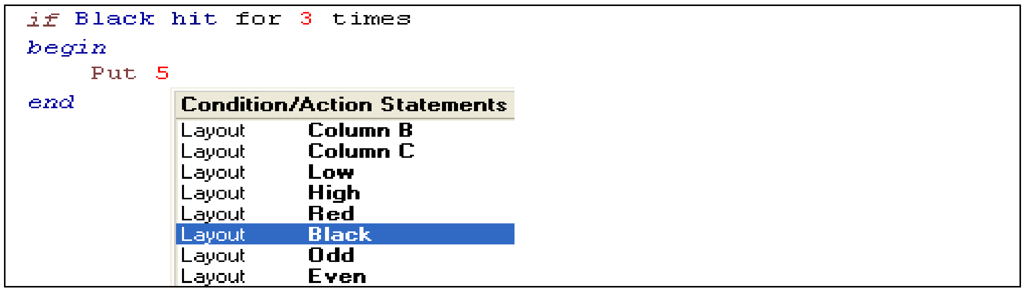
```
if Black hit for 3 times
begin
    |
end
```

2. After pressing the enter key, the word **Put** is added to your system. Now again press the **F2** key. This will display the next set of valid identifiers. Locate and select the **numeric data** identifier, then press the enter key. The system will insert a **1** next to the **Put** action command. At this point, manually change the numeric value from **1** to **5**.



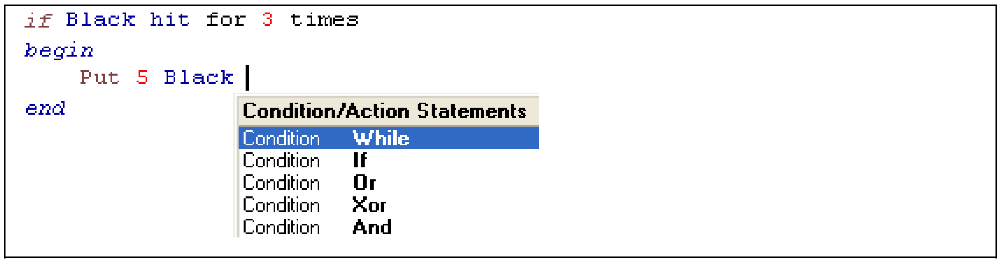
```
if Black hit for 3 times
begin
    Put |
end
```

3. Press the **F2** key. This will display the next set of valid identifiers. Locate and select the layout **Black**, then press the enter key.



```
if Black hit for 3 times
begin
    Put 5
end
```

4. The identifier **Black** will be added to your system. Press the **F2** key. This will display a complete list of condition and action commands which denote the end of the **Put** action statement.



```
if Black hit for 3 times
begin
    Put 5 Black
end
```

5. Now complete the sentence to make it easier to read by adding the non-filler words: **units on**.

```
System “my system”

Method “main"
Begin
     If Black has Hit for 3 times
     Begin
        Put 5 units on Black
     End
End
```

As you can see, using the built-in statement completion makes it very easy to construct any valid action or condition statements.

The Initial System Design

For each system design to be valid, there are two required keywords that must exist for the system to run correctly.

System

The keyword **System** must be located at the first line in the document. Following this keyword there would be a short text description enclosed in quotation marks **"**.

```
System "My first system"
```

The keyword **System** tells the system compiler that this is a valid Roulette Xtreme system.

Method

The keyword **Method** is a control block enclosed by the identifiers **Begin** and **End**. Inside the control block is where all action and condition statements are performed. Methods help group different routines of your system together. For example:

```
System "My first system"

Method "main"
Begin

End

Method "place bets"
Begin
   Put 1 unit on List [13, 21, 32]
End
```

The above method called **place bets** is created once to place bets on numbers 13, 21 and 32. The following script demonstrates how the system can call the same method more than once without duplicating the same statements by using the action command **Call**.

```
System "my system"

Method "main"
Begin
     If Black has Hit for 3 times
     Begin
        Call "place bets"
     End

     If Number 0 has Hit Each time
     Begin
        Call "place bets"
     End
End

Method "place bets"
Begin
   Put 1 unit on List [13, 21, 32]
End
```

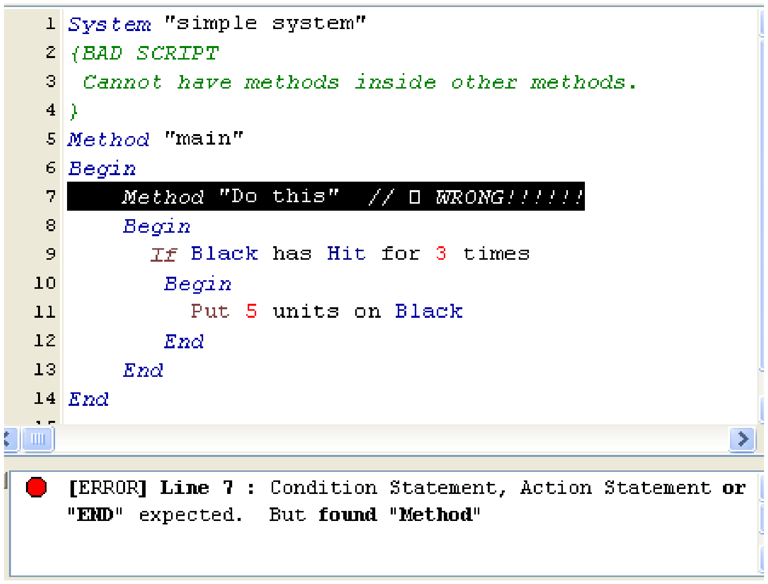
Although you can create many methods (or none if you so choose), however you must have one method called **main** as noted in the next script example. Without this method, the system will not be able to find its entry point generate an error message causing the system not to operate.

```
System “my system”

Method “main"
Begin
     If Black has Hit for 3 times
     Begin
        Put 5 units on Black
     End
End
```

When you create multiple methods, they must be separated from each other. In other words, you cannot create methods inside other methods. The scrip example below shows a bad way to use multiple methods:

```
System “my system”
{BAD SCRIPT
 Cannot have methods inside other methods.
}
Method “main"
Begin
    Method “Do this”  // ← WRONG!!!!!!
    Begin
      If Black has Hit for 3 times
       Begin
         Put 5 units on Black
       End
    End
End
```

The image below demonstrates the error message when trying to create methods within methods.



```
System "simple system"
{BAD SCRIPT
 Cannot have methods inside other methods.
}
Method "main"
Begin
    Method "Do this"   // □ WRONG!!!!!!
    Begin
      If Black has Hit for 3 times
       Begin
         Put 5 units on Black
       End
    End
End
```

[ERROR] Line 7 : Condition Statement, Action Statement or "END" expected. But found "Method"

Block Structure

A block structure is a section of the program that encapsulates all of the statements that will run inside the block. A block structure is used for all methods and logical condition statements. The block is identified by two important keywords.

- **Begin**: this keyword that is followed by a method block or condition block and denotes the start of a block. When the program enters a block structure, its entry point is just passed the **Begin** identifier.

- **End**: this keyword that denotes the end of the block structure. Once the program reaches the **End** keyword, the program returns back to the calling statement. If the block was the method **main** then the program will exit and wait for another spin to be processed either manually or automatically. If the block was a condition statement, the program will continue forward to the next statement.

The identifiers **Begin** and **End** work in pairs meaning for every **Begin** identifier; there must be an **End** identifier. The system is designed to allow you to have many block structures to help improve the functionality and ease of readability of your designed system. The example below shows how the **Begin** and **End** identifiers are placed just after a **Method** or condition statement to form a block.

```
System “my system”

Method “main"
Begin
     If Black has Hit for 3 times
     Begin
        Call “place bets”
     End
End

Method “place bets”
Begin
   Put 5 units on the 1st Dozen
End
```

Format Structure

The format of the system language within the system editor is loose, meaning that the language identifiers do not have to be located at certain lines or spaces. The compiler is smart enough to know when a statement begins and ends. The example below shows how program statements can be scattered on multiple lines. As long as the statement is in the correct order, you can place them on separate lines. This can be helpful with long statements that must wrap to the next line.

```
System “my system”

Method
"main"
Begin If Black has
  Hit Each time Begin
     Put 5 units
 on
    Black
End End
```

To help make your system easy to read and understand, it is good practice to format your system in a structured format. Always place methods on a single line followed by the **Begin** identifier on the next line at column 1. If possible place condition statements on one line followed by the **Begin** identifier just below the initial part of the condition identifier on the next line. The same is true for any action statements. If the condition or action statement cannot fit on one line, then place the remaining statement on the next line indented by three spaces. When placing the **End** identifiers, always place them inline with its partner **Begin** identifier in the same column location. A good format is shown below.

```
System “my system”

Method “main"
Begin
   If Black has Hit Each time
   Begin
      Put 5 units on Black
   End
End
```

You can also place the **Begin** identifier just after the condition statement on the same line and place the **End** identifier inline with the initial part of the condition identifier. This type of format is used by many system designers that I have seen on some roulette boards. An example of this is shown below.

```
System “my system”

Method “main"
Begin
   If Black has Hit Each time Begin
      Put 5 units on Black
   End
End
```

Using Comments

Comment identifiers are used for documenting your system. You can place comments anywhere in your system and it is considered best practice. Comments help explain what the system is trying to accomplish. Without comments, it may be difficult to understand the logic of the system. You can add comments two different ways:

- **Multi-line**: documented text information that is enclosed in brackets **{ }**. You can use this when documenting a section of the designed roulette system that spans more than one line.
- **Single-line**: documented text information that is located after two forward slashes ***//***. You can use this when documenting a single comment line.

The next example shows the two different types of comments.

```
System “my system”
{
This simple system waits until the color black has repeated
three times in a row.  Once this occurs, a 5 unit bet will be
placed on the color black.
}
Method “main" //main entry point
Begin
// Check if black repeats 3 times
     If Black has Hit for 3 times
     Begin
        { Black repeated 3 times
          in a row. Place 5 units
          on black
        }
        Put 5 units on Black
       //end of condition block
     End
{end of main method.
return back to main program
}
End
```

The more comments added to your system, the easier it is to understand your system.

Storing Data Values

The power of RX Scripting is the ability to store data values. Data values consist of logical values that consist of a true or false condition or storage values such as your beginning bankroll balance, betting amounts or roulette layouts. Storing data values are easy to do and Roulette Xtreme provides an easy way to view data values that have been stored. The two ways to store data values are using what's called **data records** and **data flags**.

Data Records

Data records provide a way to store numeric data, (i.e. 5, 4.1, 100) and roulette layouts, (i.e. black, 1st Dozen, number 19). Once data is stored in data records, they can be retrieved to be used in a variety of different ways. Here are some examples of their usage.

- Store a list of progression numbers when placing bets, (i.e. 1, 2, 4, 8, 16, 32)
- Store a list of roulette layouts to place bets and use for comparisons (i.e. number 1, number 19)
- Store a counter to keep track of wins and losses or point to the next progression in a list

The syntax for storing values to a data record is: **Record** "some name". Note the "some name" text information. Every data record must have a unique name enclosed in quotation marks in order to reference the data record within your system. When storing numeric data to your system, you will use the identifier **Data** and when storing roulette layout information, you will use the identifier **Layout**. These are both added to the end of the **Record** "some name". There are additional identifiers used with data records as well but I'll discuss those later in this book. The following example shows the usage of data records of storing numeric and layout values. Note the identifiers **Data** and **Layout** at the end of the statement.

```
System "my system"

Method "main"
Begin
   // Initialize on a new session
   While Starting a New Session
   Begin
      Set List [1, 2, 4, 8] to Record "progression" Data

      Copy List [21, 0, 19, 13, 32] to
          Record "roulette numbers" Layout

      Set Flag "ready to bet" to False
    End
End
```

Data Flags

Data flags provide a way to store logical information such as **true** or **false**. Within your designed system, there may be times when you need to perform some action based on a logical condition using a logical value that was previously stored. Here are some examples:

- A logical value to indicate when to end a session.
- A logical value to indicate when to start placing bets

The syntax for storing values to a data flag is: **Flag** “some name” followed by the identifier **True** or **False**. Note the “some name” text information. Every data flag must have a unique name enclosed in quotation marks in order to reference the data flag within your system. The following example shows the usage of data flags of storing logical values.

```
System “my system”

Method “main"
Begin
   // Initialize on a new session
   While Starting a New Session
   Begin
      Set List [1, 2, 4, 8] to Record “progression” Data

      Copy List [21, 0, 19, 13, 32] to
          Record “roulette numbers” Layout

      Set Flag “ready to bet” to False
    End
End
```

It is always good practice to identify the data records and data flags by using text names that make sense to the type of data being stored. For example, if you want to create a list of progression bets, name your data record as **progression.** Also, if you need to store a true/false value and use that to indicate when to place bets, then use a data flag and name it as **place bets**. This will let you know the data flags purpose. So, when this data flag is set to true, you will know when to place bets and perform the necessary action statements to accomplish this.

Logical Comparisons for Condition Statements

When using a condition statement, the system performs some type of logical comparison that returns either true or false. Based on the results, you can perform some action or another conditional process. You can perform a comparison from one data record to another or perform a comparison of a roulette number to a numeric value (i.e. check to see if a particular layout for example, black has appeared *n* number of times).

The following several tables lists all of the possible logical comparisons that are supported and their usage. They are group by their comparison function. The table lists four variables such as **X**, **Y**, ***n*** and ***t*** to denote some property. Their descriptions are listed below.

- **X**: represents the left side of the comparison statement. Could be a roulette layout such as Number 21, Black, Column A or an internal variable such as Bankroll or a data record.
- **Y**: represents the right side of the comparison statement. Could be a roulette layout such as Number 21, Black, Column A or an internal variable such as Bankroll or a data record.
- ***n***: represents a numeric value
- ***t***: represents a second numeric value (if more than 1 numeric value is required) for this type of comparison

Logical expression comparisons

Comparison	**What it does**
X = Y	Returns true if X and Y are equal
X Not = Y	Returns true if X and Y are NOT equal
X > Y	Returns true if X is greater than Y
X >= Y	Returns true if X is greater than or equal than Y
X < Y	Returns true if X is less than Y
X <= Y	Returns true if X is less than or equal to Y

Below is an example of using the logical comparison of **=** (equal) to check if the bankroll is equal to 15. As noted in the previous table, the X variable is represented by the identifier **Bankroll** and the Y variable is represented by the numeric value **115**.

```
System “my system”

Method “main"
Begin
   If Bankroll = 115   // noted as: X = Y
   Begin
       Stop Session
   End
End
```

Count of total comparisons

Comparison	What it does
X count = *n*	Returns true if total count of X equals *n*
X count NOT = *n*	Returns true if total count of X is NOT equal to *n*
X count > *n*	Returns true if total count of X is greater than *n*
X count >= *n*	Returns true if total count of X is greater than or equal to *n*
X count < *n*	Returns true if total count of X is less than *n*
X count <= *n*	Returns true if total count of X is less than or equal to *n*

Below is an example of using the **Count =** (equal) comparison to check if the total count of individual bets placed on the roulette table is equal to 3. As noted in the previous table, the X variable is represented by the identifier **Total Number Bets** and the *n* variable is represented by the numeric value **3**.

```
System “my system”

Method “main"
Begin
   If Total Number Bets Count = 3   // X count = n
   Begin
     Put 2 units on Number 0
   End
End
```

Lost bets comparisons

Comparison	What it does
X lost *n*	Returns true if X has lost for *n* times in a consecutive row
X lost each	Returns true if X has lost each time
X lost more *n*	Returns true if X has lost more than *n* times in a consecutive row

Below is an example of using the **Lost** comparison to check if the even chance of black has lost a bet 2 times. As noted in the previous table, the X variable is represented by the identifier **Black** and the *n* variable is represented by the numeric value **2**.

```
System “my system”

Method “main"
Begin
   If Black has Lost 2 times in a row
   Begin
      Add 10 units to Black
   End
End
```

Won bets comparisons

Comparison	What it does
X won *n*	Returns true if X has won for *n* times in a consecutive row
X won each	Returns true if X has won each time
X won more *n*	Returns true if X has won more than *n* times in a consecutive row

Below is an example of using the **Won** comparison to check if the even chance of red has won a bet 3 times. As noted in the previous table, the X variable is represented by the identifier **Red** and the *n* variable is represented by the numeric value **3**.

```
System “my system”

Method “main"
Begin
   If Red has Won 3 times in a row
   Begin
      Add 10 units to Black
   End
End
```

Outcome of roulette layout comparisons

Comparison	What it does
X hit *n*	Returns true if X has appeared for *n* times
X not hit *n*	Returns true if X has NOT appeared for *n* times
X hit each	Returns true if X has appeared each time
X NOT hit each	Returns true if X has NOT appeared each time
X hit more n	Returns true if X has appeared more than *n* times
X NOT hit more n	Returns true if X has NOT appeared more than *n* times
X hit between *n t*	Returns true if X has appeared between *n* and *t* times
X NOT hit between *n t*	Returns true if X has NOT appeared between *n* and *t* times

Below is an example of using the **Hit** comparison to check if the 1st dozen has appeared for 5 times in a row. As noted in the previous table, the X variable is represented by the identifier **1st Dozen** and the *n* variable is represented by the numeric value **5**.

```
System “my system”

Method “main"
Begin
   If 1st Dozen has Hit for 5 times in a row
   Begin
      Put 10 units on 2nd Dozen
      Put 10 units on 3rd Dozen
   End
End
```

Found within a list of numbers comparisons

Comparison	What it does
X found Y	Returns true if X is found within a list of Y values
X NOT found Y	Returns true if X is NOT found within a list of Y values

Below is an example of using the **Found** comparison to check if the last number that has appeared is within a list of 15 repeated numbers. As noted in the previous table, the X variable is represented by the identifier **Record** "last number" **Layout** and the Y variable is represented by the data record of **Record** "last 15 numbers" **Layout**. The comparison would return a true result if the roulette number stored in the data record **last number** is found within a list of numbers stored in data record **last 15 numbers**.

```
System “my system”

Method “main"
Begin
   Copy Last Number to Record "last number" Layout

   If Record "last number" Layout is Found within
      Record "last 15 numbers" Layout
   Begin
      Put 1 unit bet on Record "last 15 numbers" layout List
   End

   Track last Number for 15 spins to
      Record "last 15 numbers" layout
End
```

Last Answer comparisons

Comparison	What it does
Last answer Yes	Returns true if the last input answer is equal to Yes
Last answer No	Returns true if the last input answer is equal to No

Below is an example of using the **Last Answer** comparison to check the value of the last answer variable contains either a **Yes** or **No** value. The last answer variable is assigned by the **Ask** dialog input command.

```
System “my system”

Method “main"
Begin
   If Bankroll < 100
   Begin
      Ask "Do want to play another session?"

      If the Last Answer is No then
      Begin
        Stop Session
      End
   End
End
```

Span between 2 numbers on roulette wheel comparison

Comparison	What it does
X span *n* Y	Returns true if the roulette number X is between the distance of roulette number Y by *n* on the roulette wheel

Below is an example of using the **Span** comparison to check if roulette number X is *n* gaps between the roulette number Y. As noted in the previous table, the X variable is represented by the identifier **Record** “number 1” **Layout** and the Y variable is represented by the data record of **Record** “number 2” **Layout** and *n* variable is represented by the numeric value **4**. The comparison would return a true result if the roulette number stored in data record **number 1** is 4 gaps between the roulette number stored in data record **number 2** on the roulette wheel.

```
System "my system"

Method “main"
Begin
   Locate Number Backward 1 spin from Last Number
      to Record "number 2" Layout

   Copy Last Number to Record "number 1" Layout

   If Record "number 1" Layout Span is within 4 gaps
      from Record "number 2" Layout
   Begin
      Put 5 units on 1st Dozen
   End
End
```

Distance between 2 numbers on the marquee board comparison

Comparison	What it does
X distance *n* Y	Returns true if the distance of *n* is between X and Y

Below is an example of using the **Distance** comparison to check if the roulette number X is *n* distance between the roulette number Y. As noted in the previous table, the X variable is represented by the identifier **Record** “number 1” **Layout** and the Y variable is represented by the data record of **Record** “number 2” **Layout** and *n* variable is represented by the numeric value **5**. The comparison would return a true result if the roulette number stored in data record **number 1** distance is with 5 outcomes from the roulette number stored in data record **number 2** on the roulette wheel.

```
System "my system"

Method “main"
Begin
   If Record "number 1" Layout Distance is within 5
      spins from Record "number 2" Layout
   Begin
      Put 2 units on Record "number 1" Layout
      Put 2 units on Record "number 2" Layout
   End
End
```

Pattern Match comparisons

Comparison	What it does
X pattern match Y	Returns true if a list of values in X has the exact pattern sequence of a list of values in Y
X pattern NOT match Y	Returns true if a list of values in X does not have the exact pattern sequence of a list of values in Y

Below is an example of using the **Pattern Match** comparison to check if the previous even chance outcomes produced a pattern of black, black, red which is compared to the contents of the data record **patterns**. As noted in the previous table, the X variable is represented by the identifier **List [**Black, Black, Red**]** and the Y variable is represented by the data record of **Record** “patterns” **Layout**. The comparison would return a true result if the contents of the data record **patterns** contain a list of even chance layouts of black, black, red.

```
System “my system”

Method “main"
Begin
   Track last Red-Black patterns for 3 spins to
      Record "patterns" layout

   If List [Black, Black, Red] has a Pattern Match to
       Record "patterns" Layout
   Begin
      Put 5 units on Red
   End
End
```

WORKING WITH CONDITION COMMANDS

2

Before I discuss how to place bets using action commands, I figure it would be best to discuss condition commands, what are they and how they are used? Since most systems use some type of condition response, it would be best to learn these first.

Condition statements are commands that form a condition block. The logical outcome will produce either a true or false result. If the result is true, the condition block will execute. Some conditions could be starting a new session or testing to see if the roulette even chance of black has appeared 5 times in a row. The first part of a condition statement is a reserved word that initiates the condition followed by 1 or several supporting identifiers to complete the statement.

The system language can support multiple condition statements to form a complex condition block. Three special condition identifiers, **Or**, **And**, and **XOR** are used to concatenate multiple conditions together. If the outcome of the entire multiple condition blocks becomes true, the condition block will execute.

So to begin, I will discuss the three types of condition commands and their usage.

- Initial Condition
- Continuation Condition
- Block Condition

Initial Condition

To create a condition statement, you start with the initial condition. The following table lists the various initial condition commands that are used when creating a conditional event within your system.

Initial	**Their meaning**
While	Initial process of a logical condition statement Example: While A = B, do this….
If	Same effect as the **While** condition command Example: If A = B, do this….

The following script uses the initial condition command **While**,

```
System “my system”

Method “main"
Begin
   While Starting a New Session
   Begin
      Set List [1, 2, 4, 8, 16] to Record “progression” Data
      Put 1 unit on Red
   End
End
```

While or **If** condition?

From the previous example, I used the initial command **While** to test if a new session has started. Another conditional command **If** is exactly the same type of condition as noted in the following example.

```
System “my system”

Method “main"
Begin
   If Starting a New Session
   Begin
      Set List [1, 2, 4, 8, 16] to Record “progression” Data
      Put 1 unit on Red
   End
End
```

So, you may ask which one do I used.

When Roulette Xtreme was first created, the only initial command that existed was the **While** command. Later, the condition command **If** was added to allow those who like to use this command since it is similar to a Visual Basic syntax. It is basically just a matter of preference on which one to use.

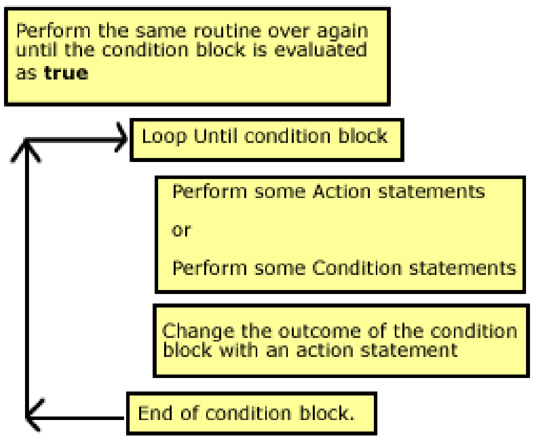
Loop Until condition

The **Loop Until** condition command is similar to the **While** or **If** commands except for two items:

- If the outcome is evaluated as **false**, the program control is passed to the body of the condition block.
- Once all of the action or condition commands within this conditional block have been performed, program control is always return back to the beginning of the condition block and the entire process is repeated until the outcome is evaluated as **true** which then ends this condition.

The following table lists the **Loop Until** and its meaning along with a diagram of the loop until process.

Initial	Their meaning
Loop Until	A condition block that processes statements in a closed loop until some event causes the loop to stop Loop Until A > B


Perform the same routine over again until the condition block is evaluated as true
Loop Until condition block
Perform some Action statements
or
Perform some Condition statements
Change the outcome of the condition block with an action statement
End of condition block.

Every time the logical event is false, all of the commands within this block are performed, and then repeated again until the event becomes true. At this point the condition block ends and the program continues just after the **End** identifier. For those who are programmers, this **Loop Until** command is similar to a **Do Until** syntax in Visual Basic. For example, suppose you had a program to add a series of numbers, but you never wanted the sum of the numbers to be more than 100. For the Visual Basic syntax, the following code would look like this:

```
Dim sum As Integer = 0

Do Until sum >= 100
   sum = sum + 10
Loop
```

And from the example above, for RX Scripting, the following code would look like this:

```
Put 0 on Record "sum" Data

Loop Until Record "sum" Data >= 100
Begin
    Add 10 to Record "sum" Data
End
```

In the above RX Scripting code, the **Loop Until** line evaluates the data record **sum** to see whether it is less than 100: If it is, the next line of RX Scripting action command is run; if not, it moves to the next line of code following **End** identifier. The **End** identifier tells RX Scripting to go back to the **Loop Until** line and evaluate the new value of the data record **sum**.

Continuation Condition

Sometimes it is necessary to create a conditional event that occurs when two or more logical events may occur. For this to happen, you concatenate these conditional commands together using one of the three continuation commands: **And**, **Or** and **Xor**.

The following table lists these continuation commands, their meaning and their truth table.

Cond.	Their meaning	Truth table		
		A	**B**	**A and B**
And	Known as a logical conjunction in which the outcome result is true if **all** of the combined condition statements are true; else the outcome is false. While A = B **AND** C < D, do this…	False	False	False
		False	True	False
		True	False	False
		True	True	True
		A	**B**	**A and B**
Or	Known as a logical disjunction in which the outcome result is true if **any** of the combined condition statements are true; else the outcome is false. While A = B **OR** C < D, do this…	False	False	False
		False	True	True
		True	False	True
		True	True	True
		A	**B**	**A and B**
Xor	Known as an exclusive disjunction in which the outcome result is true if **either** of the combined condition statements is true but not both; else the outcome is false. While A = B **XOR** C < D, do this…	False	False	False
		False	True	True
		True	False	True
		True	True	False

The example below is showing how to concatenate the different continuation condition commands.

```
System “my system”

Method “main"
Begin
   If  Black has Hit 3 times in a row
   And High has Hit 2 times in a row
   Or  1st Dozen has Hit 1 time in a row
   Begin
      Put 5 units on Column A
      Put 5 units on Column C
   End
End
```

Note the example is checking for three different events. By using a standard truth table (as shown in the previous table), one could see that the statement above reads like this:

- When black has appeared for 3 times **And** high numbers have appeared for 2 times, place 5 units on columns A and C

Or

- When the 1st dozen has appeared for 1 time, place 5 units on columns A and C.

When combining logical **And** identifiers with **Or** and **Xor** identifiers together to create a condition block, the system will create implied parentheses **( )** around the **And** identifier. For example,

the condition statement format:

- If condition 1
  And condition 2
  Xor condition 3
  And condition 4
  Or condition 5
  then…

will be evaluated as

- If **(**condition 1 **And** condition 2**)**
  Xor
  (condition 3 **And** condition 4**)**
  Or
  (condition 5**)**
  then…

All of the **And** identifiers are evaluated first, then the results will be evaluated with the **Or** and **Xor** identifiers to determine the final outcome.

Using the Else continuation

The **Else** identifier is used in conjunction with either **While** or **If** initial condition commands and therefore cannot be used as a standalone command.

The following table lists the **Else** identifier and its meaning.

Continuation	Their meaning

Else	This condition is used with either the While or If condition at the end of the condition block. It causes the program to execute statements if the initial condition statement returns false.
	
	While A = B
	begin
	   do this…
	end
	Else
	begin
	   do this one..
	end

Whenever the initial condition command is evaluated as false, the system will execute lines after the **Begin** identifier from the **Else** command. For example:

```
System “my system”

Method “main"
Begin
   If Black has Lost Each time
   Begin
      Put 5 units on Red
   End
   Else
   Begin
      Put 5 units on Black
   End
End
```

If the even chance layout of black had won a bet on each spin, the outcome of the condition statement would be false (since we are looking for a loss and not a win), and the program would move to the **Else** command and execute all lines following the **Begin** identifier. In this example, the system would then place a 5 unit bet on the even chance of black.

.

Block Condition

Using the Group Condition

The block condition command is a special command mainly used for grouping user dialog information together. The following table lists the **Group** identifier and its meaning.

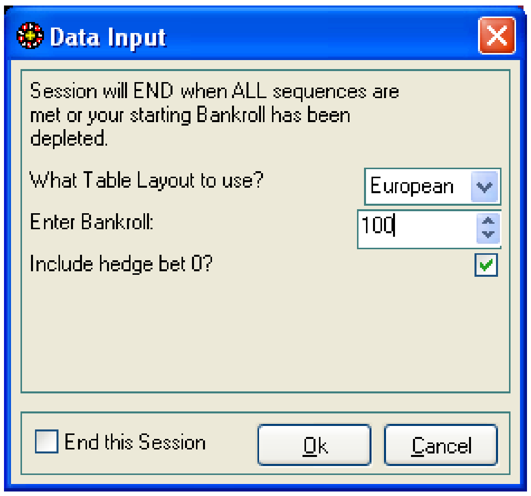
Block	Their meaning
Group	This condition is always evaluated as true and therefore, allowing special action commands to execute within this block.

By using the block condition, the user will be presented with an input dialog screen containing one or many different input controls instead of having the system present them one at a time.

When you use the **Group** condition command, only certain action commands and not condition commands are allowed. Otherwise the system will generate an error message. Below is a list of the allowable action commands.

Allowable Action commands	Their usage…
Display *message*	Displays information from the *message* to the screen
Input Data *message* Y	Prompts user with *message* for input of a numeric value into Y.
Input Dropdown *message* Y	Prompts user with *message* to select an item for input from a drop down list of values into Y.
Input Checkbox *message* Y	Prompts user with *message* for input to select a checkbox.

The screen shot below shows how the **Group** condition block with the allowable action commands is presented to the user. Note each of these commands creates an active user interface thus providing feedback to enter information before or during a session. The only exception to this is the **Display** action command which only provides information to the user. The RX Scripting for this example is on the next page.


Data Input

Session will END when ALL sequences are met or your starting Bankroll has been depleted.

What Table Layout to use? European

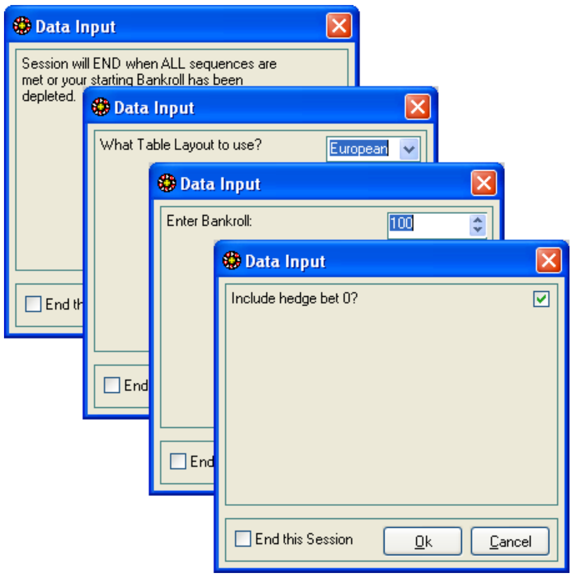
Enter Bankroll: 100

Include hedge bet 0? ✓

RX Scripting with all allowable action commands within a **Group** condition command.

```
System "my system"

Method "main"
Begin
   // Initialize on a new session
   While Starting a New Session
   Begin
      Group
      Begin
         Display
            "Session will END when ALL sequences are
             met or your starting Bankroll has been
             depleted."

         Input Dropdown
            "What Table Layout to use?
             1:=European
             2:=American" to Record "table" Data

          Input Data
             "Enter Bankroll:" to Record "bankroll" Data

          Input Checkbox
             "Include hedge bet 0?" to Flag "hedge bet"
       End
    End
End
```

If you were to omit the **Group** condition command from script above, the system will present each of the action commands to the user one screen at a time as noted in the next screen shot example.


Session will END when ALL sequences are met or your starting Bankroll has been depleted.

What Table Layout to use? European

Enter Bankroll: 100

Include hedge bet 0? ✓

Sometimes you may want this to happen during a session for example, to let the user know when a session has ended.

WORKING WITH ACTION COMMANDS 3

Now that you have learned condition commands and how they work with designing systems it is now time to know how to write action commands like placing bets, storing data into data records and such.

Action statements are commands that perform some immediate action without any regards to the outcome of a logical condition. Some actions could be placing bets on the roulette layout or storing data values into a data record. The first part of an action statement is a reserved word that initiates the action followed by supporting identifiers to complete the action statement.

In this chapter, I will discuss the five types of action commands and their usage.

- Place Action
- Mathematical Action
- Input Action
- Assign Action
- Maneuver Action

Place Action

Place action commands allow you to place values onto roulette layouts, assign values to internal variables or data records. The next set of action commands are used to assign values to various destinations. See **Table 3.0** in this chapter for the entire possible destinations.

Put

The **Put** action command has two purposes:

1. Place bets onto any roulette layout such as Black, Number 12, Split 5-8 and so on.
2. Assign values to data records or internal variables such as Bankroll.

The format is: **Put N D** where **N** is the numeric value (i.e. 1, 5, 10, 100, etc) and **D** is the destination (i.e. Black, Number 30, Record "record 1" Data, etc.) as shown in **Table 3.0**.

As a comparison, the **Put** action command is similar to an assignment function in Visual Basic. Below is an example of assigning values in Visual Basic similar to RX Scripting:

```
'Roulette Layout Black
Dim BlackLayout As Object
'data record "Amount to bet"
Dim AmountToBet As Decimal
'internal variable Bankroll
Dim Bankroll    As Decimal
'Assigment function – similar to the Put action command
BlackLayout = 5
AmountToBet = 5
Bankroll    = 100
```

And now written with RX Scripting:

```
System "my system"

Method "main"
Begin
    Put 5 units on Black
    Put 5 to Record "Amount to bet" Data
    Put 100 units to Bankroll
End
```

You may ask *What if you want to assign values stored as variables.* Use the **Put 100%** action command instead as discuss in the next section.

Put 100%

As with the **Put** action command, this command performs the same function with some exceptions:

- The value is from either a numeric value stored in a data record or the numeric value from roulette layouts or internal variables.
- That value is then multiplied by the *n* % (percent) value.
- The multiplied result is assigned to data records or internal variables such as Bankroll.

The format is: **Put 100%** **S D** where **S** is the source data value stored in a data record; the data value must be numeric (i.e. 1, 5, 10, 100, etc) and **D** is the destination (i.e. Black, Number 30, Record "record 1" Data, etc.) as shown in **Table 3.0**. The **100%** parameter can be any percent value like 100, 200, 40, 20, and so on.

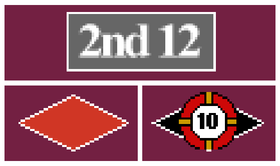
As a comparison, the **Put 100%** action command is similar to an assignment function using a multiplier symbol * and divide symbol / in Visual Basic. Below is an example assigning values in Visual Basic using the multiplier:

```
'Roulette Layout Black
Dim BlackLayout As Object
'data record "Amount to bet"
Dim AmountToBet As Decimal = 5

'put 200% of AmountToBet to Black Layout
BlackLayout = (AmountToBet * 200) / 100
```

As written in RX Scripting:

```
System "my system"

Method "main"
Begin
    Put 5 to Record "Amount to bet" Data
    Put 200% of Record "Amount to bet" Data to Black
End
```

The result of this action command is shown below. Notice that the even chance of black now has 10 units instead of 5 since the **Put 200%** doubled the amount from the value stored in the data record **Amount to bet**.



The beauty of RX Scripting is that you don't have to declare any variables. RX will do that for you. All you have to do is use the **Put** action command to place a bet or store a data value. And with the English type language, you don't have to be a programmer to perform this action.

Mathematical Action

Mathematical action commands are similar to the **Put** action commands except for the following:

- The system will perform a mathematical operation of either add, subtract, multiply or divide of a value to the value stored at a destination such as roulette layout, data record or internal variable.

Table 3.0 in this chapter is a table showing the entire possible storage destinations and their meaning when using the mathematical action commands

Add

The **Add** action command has two purposes:

1. Add bets to an existing value onto any roulette layout such as Black, Number 12, Split 5-8 and so on.
2. Add values to an existing value located at data records or internal variables such as Bankroll.

The format is: **Add** **N** **D** where **N** is the numeric value (i.e. 1, 5, 10, 100, etc) and **D** is the destination (i.e. Black, Number 30, Record "record 1" Data, etc.) as shown in **Table 3.0**.

In comparison, the **Add** action command is similar to a mathematical addition operation in Visual Basic. Below is an example of performing an add operation in Visual Basic similar to RX Scripting:

```
'Roulette Layout
Dim BlackLayout As Object
'place a bet on Black layout
BlackLayout = 5
'Add 10 to Black layout. Results = 15
BlackLayout = BlackLayout + 10
```

As written in RX Scripting:

```
System “my system”

Method “main"
Begin
    Put 5 units on Black
    Add 10 to Black
End
```

Add 100%

As with the **Add** action command, this command performs the same function with some exceptions:

- The value is from either a numeric value stored in a data record or the numeric value from roulette layouts or internal variables.
- That value is then multiplied by the *n* % (percent) value.

The multiplied result is the added to the value located at the data records or internal variables such as Bankroll.

The format is: **Add 100%** **S D** where **S** is the source data value stored in a data record; the data value must be numeric (i.e. 1, 5, 10, 100, etc) and **D** is the destination (i.e. Black, Number 30, Record “record 1” Data, etc.) as shown in **Table 3.0**. The **100%** parameter can be any percent value like 100, 200, 40, 20, and so on.

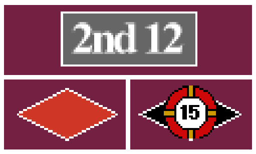
In comparison, the **Add 100%** action command is similar to an add operation function using a multiplier symbol * and divide symbol / in Visual Basic. Below is an example assigning values in Visual Basic using the multiplier:

```
'Roulette Layout
 Dim BlackLayout As Object
 'data record
 Dim AmountToBet As Decimal = 5
 BlackLayout = 5
 'Add 200% of AmountToBet to BlackLayout. Results = 15
 BlackLayout = BlackLayout + ((AmountToBet * 200) / 100)
```

As written in RX Scripting:

```
System “my system”

Method “main"
Begin
    Put 5 units on Black
    Put 5 to Record “Amount to bet” Data
    Add 200% of Record “Amount to bet” Data to Black
End
```

The result of this action command is shown below. Notice that the even chance of black now has 15 units instead of 10 since the **Add 200%** doubled the amount from the value stored in the data record **Amount to bet** and then added to the existing value in the even chance of Black.



Subtract and Subtract 100%

The **Subtract** and **Subtract 100%** action commands are the exact same functionality as the **Add** and **Add 100%** action commands except instead of performing a mathematical addition operation, they perform the mathematical subtraction operation. So, visit the **Add** section on how these commands work.

Below I will show you a brief script example in Visual Basic and RX Scripting.

```
    Dim BlackLayout As Object
    BlackLayout = 5
    'Subtract 2 from Black layout. Results = 3
    BlackLayout = BlackLayout - 2
```

As written in RX Scripting:

```
    System “my system”

    Method “main"
    Begin
        Put 5 units on Black
        Subtract 2 from Black
    End
```

Multiply

The **Multiply** action command has two purposes:

1. Multiply bets to an existing value onto any roulette layout such as Black, Number 12, Split 5-8 and so on.
2. Multiply values to an existing value located at data records or internal variables such as Bankroll.

The format is: **Multiply** **N** **D** where **N** is the numeric value (i.e. 1, 5, 10, 100, etc) and **D** is the destination (i.e. Black, Number 30, Record “record 1” Data, etc.) as shown in **Table 3.0**.

In comparison, the **Multiply** action command is similar to a mathematical multiplication operation function in Visual Basic. Below is an example of performing a multiplication operation in Visual Basic similar to RX Scripting:

```
    'Roulette Layout
    Dim BlackLayout As Object
    'place a bet on Black layout
    BlackLayout = 5
    'Multiply 2 to Black layout. Results = 10
    BlackLayout = BlackLayout * 2
```

As written in RX Scripting:

```
    System “my system”

    Method “main"
    Begin
        Put 5 units on Black
        Multiply 2 to Black
    End
```

Multiply 100%

As with the **Multiply** action command, this command performs the same function with some exceptions:

- The value is from either a numeric value stored in a data record or the numeric value from roulette layouts or internal variables.
- That value is then multiplied by the *n* % (percent) value.

The multiplied result is the multiplied to the value located at the data records or internal variables such as Bankroll.

The format is: **Multiply 100% S D** where **S** is the source data value stored in a data record; the data value must be numeric (i.e. 1, 5, 10, 100, etc) and **D** is the destination (i.e. Black, Number 30, Record "record 1" Data, etc.) as shown in **Table 3.0**. The **100%** parameter can be any percent value like 100, 200, 40, 20, and so on.

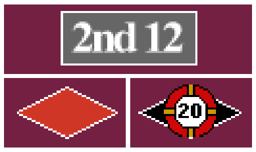
The **Multiply 100%** action command is similar to an multiply operation function using a multiplier symbol * and divide symbol / in Visual Basic. Below is an example assigning values in Visual Basic using the multiplier:

```
   'Roulette Layout
    Dim BlackLayout As Object
    'data record
    Dim AmountToBet As Decimal = 2
    BlackLayout = 5
    'Add 200% of AmountToBet to BlackLayout. Results = 20
    BlackLayout = BlackLayout * ((AmountToBet * 200) / 100)
```

The above example is written with RX Scripting on the next section:

```
System "my system"

Method "main"
Begin
    Put 5 units on Black
    Put 2 to Record "Amount to bet" Data
    Multiply 200% of Record "Amount to bet" Data to Black
End
```

The result of this action command is shown below. Notice that the even chance of black now has 20 units instead of 10 since the **Multiply 200%** doubled the amount from the value stored in the data record **Amount to bet** and then multiplied to the existing value in the even chance of black.



Divide and Divide 100%

The **Divide** and **Divide 100%** action commands are the exact same functionality as the **Multiply** and **Multiply 100%** action commands except instead of performing a mathematical multiplication operation, they perform the mathematical division operation. So, visit the **Multiply** section on how these commands work. One thing to note, the dividend is the value stored at the destination and divisor is the value stored in data record accessed by the 100% function.

Below is an example of **Divide** action command in Visual Basic and RX Scripting.

```
    'Roulette Layout
    Dim BlackLayout As Object
    ‘divide a bet on Black layout
    BlackLayout = 20
    'Divide 2 from Black layout. Results = 10
    BlackLayout = BlackLayout / 2
```

As written in RX Scripting:

```
    System “my system”

    Method “main"
    Begin
        Put 20 units on Black
        // 2 is the divisor and
        // Black is the dividend
        // similar to Black / 2 = 10
        Divide 2 from Black
    End
```

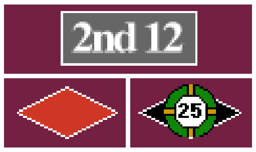
This next example is the **Divide 100%** action command in Visual Basic and RX Scripting.

```
    'Roulette Layout
    Dim BlackLayout As Object
    'data record
    Dim AmountToBet As Decimal = 2
    BlackLayout = 100
    'Divide 200% of AmountToBet to BlackLayout. Results = 25
    BlackLayout = BlackLayout / ((AmountToBet * 200) / 100)
```

As written in RX Scripting:

```
    System “my system”

    Method “main"
    Begin
        Put 2 to Record “Amount to bet” Data
        Put 100 on Black
        // Record “Amount to bet” is the divisor and
        // Black is the dividend
        // similar to Black / Record “xxx” Data = 25
        Divide 200% of Record “Amount to bet” Data to Black
    End
```

The result of this action command is shown below. Notice that the even chance of black now has 25 units instead of 50 since the **Divide** **200%** doubled the amount from the value stored in the data record **Amount to bet** and then divided to the existing value in the even chance of Black.



So you may ask, *what happens if RX divides by 0, do you get an error message?* The answer to that question is **no**. If you try to divide by zero, the system just ignores the error and returns a result of 0 to your roulette layout or data record. So this type of error message is nothing to worry about.

Table 3.0 - Storage destinations when using the place and mathematical action commands.

Identifier	**Their meaning**
Bankroll	Sets the internal variable **bankroll** from the calculated value which is displayed on the main screen under the Bankroll field.
All Bets	Places a bet from the calculated value on all of the possible bet locations on the roulette table. Any value will be place on all of these locations.
All Outside	Places a bet from the calculated value only on the outside possible bet locations on the roulette table. Any value will be place on all of these outside locations.
All Inside	Places a bet from the calculated value only on the inside possible bet locations on the roulette table. Any value will be place on all of these inside locations.
Record “xyz” **Data**	Stores a numeric value from the calculated value to the data record “xyz” data section.
Record “xyz” **Layout**	Places a bet from the calculated value to the roulette layout that is stored in this data record “xyz”.
Neighbor Count	Sets the internal variable **neighbor count** from the calculated value to be used later when working with neighbor bets.
Record “xyz” **Data Index**	Sets the data record index from the calculated value for the data section. This is used to access a number from a list of numbers stored in this data record “xyz”. (i.e. 1,2,4,6,8)
Record “xyz” **Layout Index**	Sets the data record index from the calculated value for the layout section. This is used to access a roulette layout from a list of layouts stored in this data record “xyz”. (i.e. Number, 5, Number 10, Number 13)
List [Y1,Y2,Y3**]**	Places a bet from the calculated value on all roulette layouts that are in a list enclosed in brackets. (i.e. [Number 13, Number 19, Split(5-8)])
Record “xyz” **Layout List**	Places a bet from the calculated value on all roulette layouts that are stored as a list in a data record “xyz” (i.e. Number 13, Number 19, Split(5-8))
Any roulette layout (i.e. Number 13, Corner A, etc.)	Places a bet from the calculated value directly on any of the roulette layouts. (i.e. Number 13)

Input Action

Input action commands provide an active user interface, giving you feedback to enter information before or during an active session. This is a very powerful feature because you have the ability to provide your own data to alter the outcome of your system. For example, entering a starting bet amount prior to a new session. Having your session ask you if you want to stop after reaching your win goal. The possibilities are endless on what you can do with input action commands.

The next set of action commands are used to provide an active user interface to various destinations.

Input Data

The **Input Data** action command displays an active user interface dialog which allows the user to enter a numeric value and store that value to one of several destinations such as bankroll, a roulette layout or a data record.

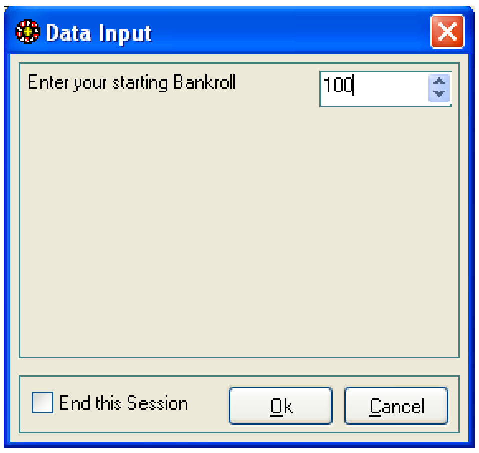
The format is: **Input Data T D** where **T** is text information enclosed in quotation marks "" and **D** the destination that is a data record, a roulette number or bankroll which contains a numeric data value. This command is written like this:

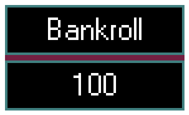
```
Input Data "Enter your bankroll" to Record "bankroll" Data
```

Here is it shown with RX Scripting within a system:

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
        Input Data "Enter your starting Bankroll"
           to Bankroll
    End
End
```

Below is the result from the **Input Data** action command with RX Scripting. After the user clicks on the Ok button, the value (in this case 100) is stored to the internal variable, **Bankroll** and then display on the main screen as noted on the right image.





Input Dropdown

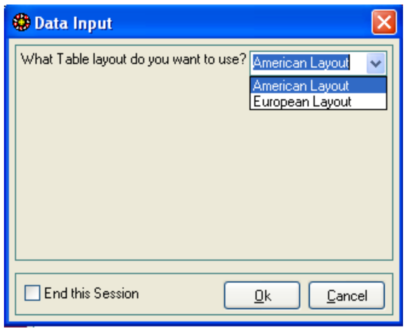
The **Input Dropdown** action command displays an active user interface dialog which allows the user to select from a list of choices and assigned its numeric index (value) to one of several destinations such as bankroll, a roulette layouts or a data record. The format is: **Input Dropdown T D** where **T** is text information enclosed in quotation marks “” and **D** the destination that is a data record, roulette layout, bankroll which contains a numeric data value. This command is written like this:

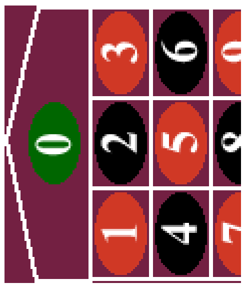
```
Input Dropdown “Choose 1 bet option
     1:=5 units
     2:=10 units
     3:=20 units” to Record “bet amount” Data
```

Note that the different dropdown selections are set by using the **n**:= symbol before the name of the selection. The **n** is the numeric value that will be stored in the data contents of a data record. Here is it shown with RX Scripting within a system.

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
       Input Dropdown
          "What Table layout do you want to use?

           1:= American Layout
           2:= European Layout"
               to Record "Table" Data

       If Record "Table" Data = 1 Begin
          Load Double Wheel
       End
       Else Begin
          Load Single Wheel
       End
    End
End
```

Below is the result from the **Input Dropdown** action command with RX Scripting from the previous section. After the user clicks on the Ok button, the numeric value of 2 (for example, if European Layout was selected) was stored in the data record **Table**. Several condition statements were performed to test the value stored in the data record **Table**. In this example if European Layout was selected from the choice list, the system loaded the single zero wheel as noted on the right image.





Display

The **Display** action command displays an active user interface dialog that displays only text information. For example, you may want to display information when your session has ended.

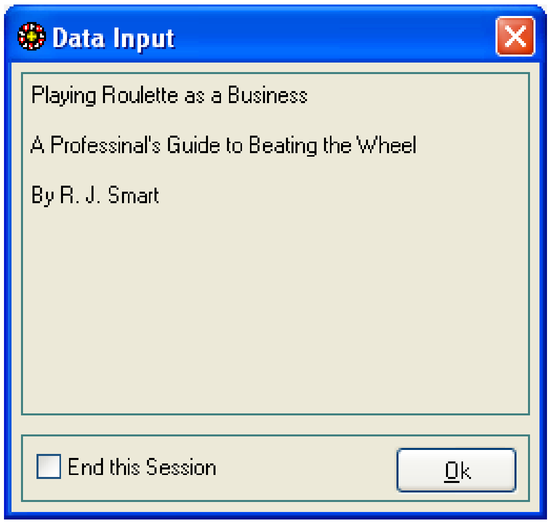
The format is: **Display T** where **T** is text information enclosed in quotation marks "". This command is written like this:

```
Display "My cool system"
```

Here is it shown with RX Scripting within a system:

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
       Display
         "Playing Roulette as a Business

         A Professional's Guide to Beating the Wheel

         By R. J. Smart"
    End
End
```

Below is the result from the **Display** action command with RX Scripting from the previous section. As noted in the script example, I am displaying text information at the start of a new session about the system that I planned on running with Roulette Xtreme. When displaying multiple lines, you type the entire text between quotation marks and place them on different lines to separate them as noted in the example below.



Playing Roulette as a Business

A Professinal's Guide to Beating the Wheel

By R. J. Smart

Ask

The **Ask** action command displays an active user interface dialog with two buttons: Yes which is **True** and No which is **False**. The result of clicking one of the two buttons will be stored in an internal variable called **answer** which then can be later referenced in a condition statement using the **Last Answer** identifier.

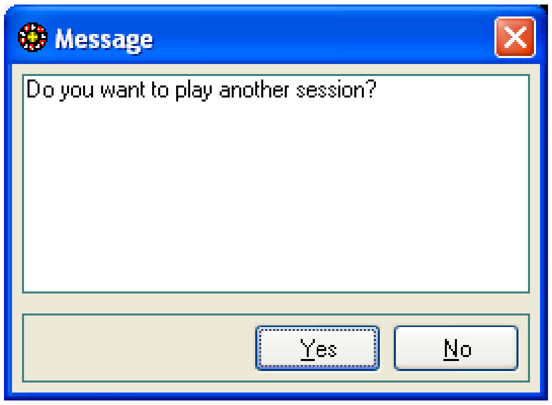
The format is: **Ask** **T** where **T** is text information enclosed in quotation marks “”.e destination. This command is written like this:

```
Ask “Do you want to quit?”
```

Here is it shown with RX Scripting within a system.

```
System “my system”

Method “main"
Begin
   If Bankroll < 100
   Begin
      Ask "Do want to play another session?"

      If the Last Answer is No then
      Begin
        Stop Session
      End
   End
End
```

Below is the result from the **Ask** action command with RX Scripting. After the user clicks either the Yes or No button, the value is stored in an internal variable called **Answer**. Later in the program, you can reference this variable by using a condition state with the **Last Answer** identifier as noted in the script example. During your session, the **Answer** variable will always contain that last Yes No selection from the **Ask** action command until either a new session is initialized or you use the **Clear Last Answer** action command.



Do you want to play another session?

Assign Action

Assign action commands provide the ability to assign values to various destinations such as a data record, copy one data record to another, capture spins that have occurred, assigned values to internal variables like Bankroll or change the way Roulette Xtreme functions like loading a table wheel and setting the **en prison** rule.

Assign action commands that alter Roulette Xtreme

The next set of action commands are used to alter Roulette Xtreme when starting a new session or during a session run.

Apply En Prison

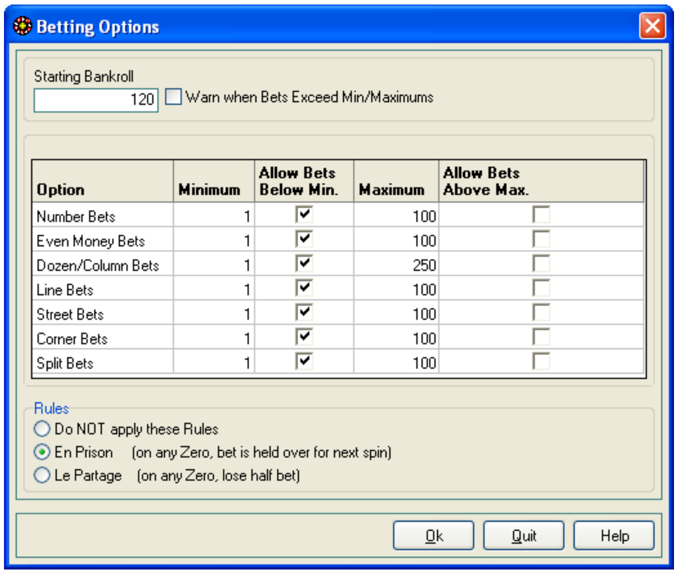
The **Apply En Prison** action command applies the **En Prison** rule. This command overrides the En Prison option that is located in the betting options screen from the Roulette Xtreme software. The En Prison rule applies mostly to the single zero wheel and lately some on-line casinos offer this for the double zero as well. So the rule will work for both wheels when using the Roulette Xtreme software.

The way this rule works is when the En Prison rule is applied, and whenever the last number that appeared is 0 (or 00 for some on-line casinos), a unit bet that has been placed on an even chance layout is held over (imprisoned) for the next spin. If the next spin is another 0 or 00, then the bet placed is lost.

The example below shows you how to code the En Prison rule with RX Scripting:

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
       Apply En Prison rule
    End
End
```

You can also manually set the En Prison rule under the Betting Options screen with the Roulette Xtreme software as noted below.



Starting Bankroll 120 [ ] Warn when Bets Exceed Min/Maximums

Option	Minimum	Allow Bets Below Min.	Maximum	Allow Bets Above Max.
Number Bets	1	✓	100	
Even Money Bets	1	✓	100	
Dozen/Column Bets	1	✓	250	
Line Bets	1	✓	100	
Street Bets	1	✓	100	
Corner Bets	1	✓	100	
Split Bets	1	✓	100	

Rules
- Do NOT apply these Rules
- [✓ En Prison (on any Zero, bet is held over for next spin)]
- Le Partage (on any Zero, lose half bet)

Apply Le Partage

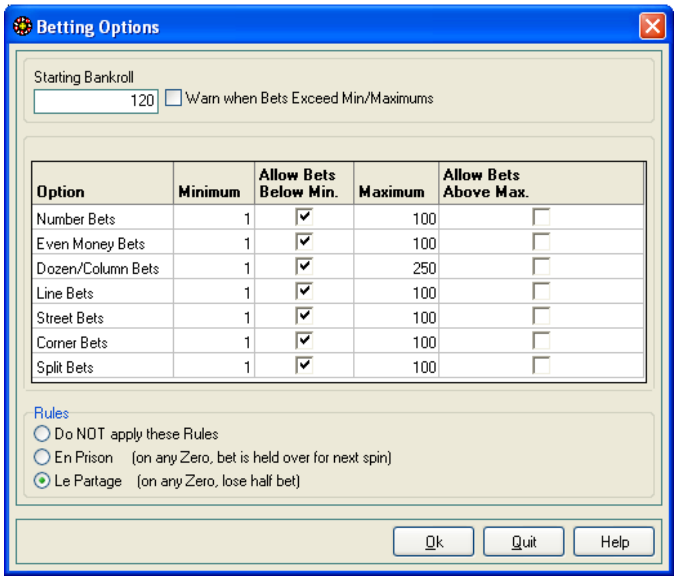
The **Apply Le Partage** action command applies the **Le Partage** rule. This command overrides the Le Partage option that is located in the betting options screen from the Roulette Xtreme software. The Le Partage rule applies mostly to the single zero wheel and lately some on-line casinos offer this for the double zero as well. So the rule will work for both wheels when using the Roulette Xtreme software.

The way this rule works is when the Le Partage rule is applied, and whenever the last number that appeared is 0 (or 00 for some on-line casinos), you will lose one-half of your unit bet that has been placed on an even chance layout is lost.

The example below shows you how to code the Le Partage rule with RX Scripting:

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
       Apply Le Partage rule
    End
End
```

You can also manually set the Le Partage rule under the Betting Options screen with the Roulette Xtreme software as noted below.



Starting Bankroll
120 ☐ Warn when Bets Exceed Min/Maximums

Option	Minimum	Allow Bets Below Min.	Maximum	Allow Bets Above Max.
Number Bets	1	✓	100	☐
Even Money Bets	1	✓	100	☐
Dozen/Column Bets	1	✓	250	☐
Line Bets	1	✓	100	☐
Street Bets	1	✓	100	☐
Corner Bets	1	✓	100	☐
Split Bets	1	✓	100	☐

Rules
- Do NOT apply these Rules
- En Prison (on any Zero, bet is held over for next spin)
- [✓ Le Partage (on any Zero, lose half bet)]

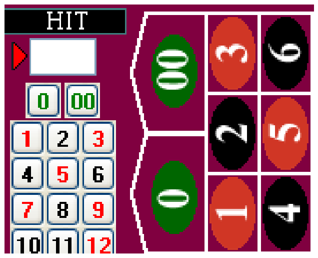
Load Double Wheel

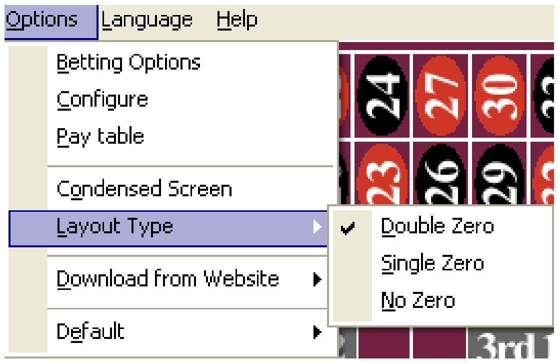
The **Load Double Wheel** action command loads the double wheel table layout in Roulette Xtreme. This command overrides the Layout Type option from the Options menu on the main screen of the Roulette Xtreme software.

The example below shows you how to code loading the double wheel table with RX Scripting:

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
       Load Double Wheel table
    End
End
```

The result from this command is shown on the left image below. The right image shows that you can also manually load the double wheel table from the Layout Type option under the Options menu from the main screen of the Roulette Xtreme software.





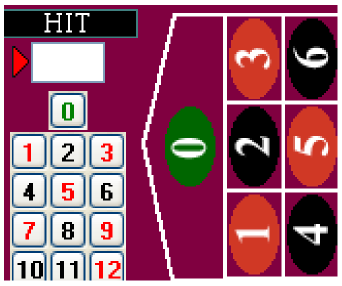
Load Single Wheel

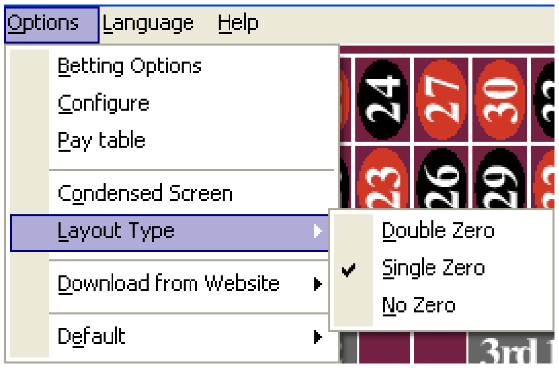
The **Load Single Wheel** action command loads the single wheel table layout in Roulette Xtreme. This command overrides the Layout Type option from the Options menu on the main screen of the Roulette Xtreme software.

The example below shows you how to code loading the single wheel table with RX Scripting:

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
       Load Single Wheel table
    End
End
```

The result from this command is shown on the left image below. The right image shows that you can also manually load the single wheel table from the Layout Type option under the Options menu from the main screen of the Roulette Xtreme software.





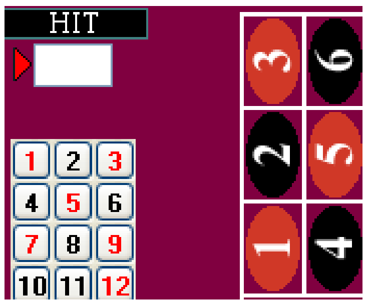
Load No Zero Wheel

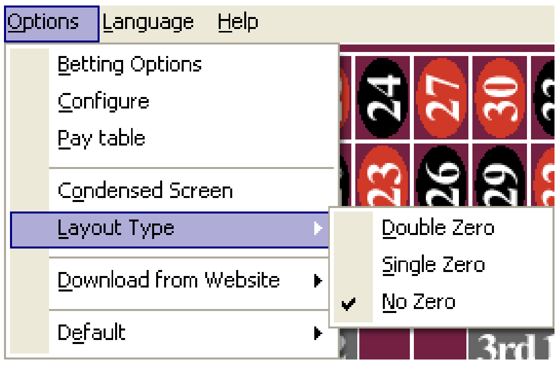
The **Load No Zero Wheel** action command loads the no-zero wheel table layout in Roulette Xtreme. This command overrides the Layout Type option from the Options menu on the main screen of the Roulette Xtreme software. There are some on-line casinos that have roulette tables without the zero.

The example below shows you how to code loading the no zero wheel table with RX Scripting:

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
       Load No Zero Wheel table
    End
End
```

The result from this command is shown on the left image below. The right image shows that you can also manually load the no-zero wheel table from the Layout Type option under the Options menu from the main screen of the Roulette Xtreme software.





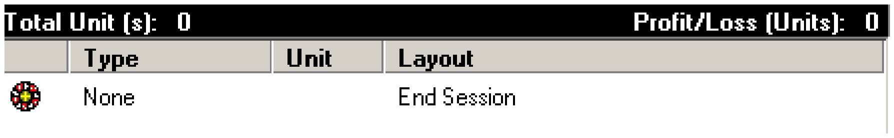
Stop Session

The **Stop Session** action command will cause the designed system to stop processing spins and halt the program. This is a great way to stop a session after you have reached a certain bankroll win or loss goal. You usually place this action command within a condition command so it will only execute when a certain condition or user prompt occurs.

The example below shows you how to code with RX Scripting. The action command is performed only after the user answer No to the **Ask** user prompt.

```
System “my system”

Method “main"
Begin
    If Bankroll < 50
    Begin
       Ask "Do want to play another session?"

       If the Last Answer is No then
       Begin
         Stop Session
       End
    End
End
```

The result from this command is shown below. Notice the word End Session under the layout column. This indicates that your system will stop processing. The only way to reset is to start a new session.



Total Unit (s): 0 Profit/Loss (Units): 0

	Type	Unit	Layout
	None		End Session

Assign action commands that copy data

There are action commands that copy roulette layouts to data records for the purpose of placing bets or analyzing patterns. One of the best examples I can give is copying the last number that has appeared or its family member (i.e. last split, last line, last dozen) to a data record. When you copy information to data records, you can then review them during your system to place bets, change progression amounts or test for certain conditions. You can also duplicate an existing data record to another data record of a different name.

Copy

This action command copies any roulette layout either directly or from a data record to a destination data record. The format is: **Copy** **S D** where **S** is the source and **D** the destination. The source **S** can either be:

- any valid roulette layout (i.e. **Number 13**, **Number 0**, **1st Dozen**, **Red**, **Odd**)
- the layout part of a data record: **Record** "source" **Layout**

The destination **D** is the layout part of a data record: **Record** "destination" **Layout**.

The following script shows some examples. The first line will copy the 1st dozen roulette layout to the layout contents of data record **dozen 1** and the second line will copy the layout contents of data record **last** to the layout contents of data record **number**.

```
Copy 1st Dozen to Record "dozen 1" Layout

Copy Record “last” Layout to Record “number” Layout
```

The following system shows an example of how to use the **Copy** action command. The 1st dozen roulette layout is copied to a data record **dozen 1** during the system initialization. For every spin, the system checks to see if the 1st Dozen stored in the data record **dozen 1** has not appeared for 3 times in a row. If this condition is true, then the system will place a 5 unit bet on the 1st dozen.

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
       Copy 1st Dozen to Record “dozen 1” Layout
    End

    While on Each Spin
    Begin
       If Record “dozen 1” Layout has Not Hit 3 times
       Begin

          Put 5 units on Record “dozen 1” Layout
       End
    End
End
```

Copy Last

This action command copies the last roulette layout or the last even chance pair that has appeared to a destination data record. The format is: **Copy Last** **S** **D** where **S** is the source and **D** the destination. The source **S** can either be:

- the last roulette layout (i.e. last **Split**, last **Number**, last **Dozen**)
- the last even chance pair (i.e. last **Even-Odd**, last **Red-Black**, last **High-Low**)

The destination **D** is the layout contents of a data record: **Record** "destination" **Layout**.

The following script shows some examples. The first line will copy the last straight-up number that has appeared to the layout contents of data record **last number** and the second line will copy the last even chance pair of red/black that has appeared to the layout contents of data record **last red/black**.

```
Copy Last Number to Record "last number" Layout

Copy Last Red-Black to Record “last red/black” Layout
```

The following system shows an example of how to use the **Copy Last** action command. The last even chance pair of red/black that has appeared is copied to the layout contents of data record **last red/black** for every spin. Then the system will place a 5 unit bet on that even chance pair which could either be red or black. This type of betting system is known as follow the last color bet.

```
System “my system”

Method “main"
Begin
    While on Each Spin
    Begin
       Copy Last Red-Black to Record "last red/black" Layout

       Put 5 units on Record "last red/black" Layout
    End
End
```

Copy List

This action command creates a list of roulette layouts to a destination data record. The purpose of this action command is to allow you to place bets on a set of roulette layouts all at once. For example: if I want to place bets on the roulette layouts Number 19, Number 32, Red, Line 1-6 and Column A, I can do that by creating a list of these layouts and assign them to the layout contents of a data record. Then by using the **Put** action command, I can place bets on all of these layouts with a single action command.

The format is: **Copy List** [**S**] **D** where [**S**] is the source and **D** the destination. The source [**S**] is a list of roulette layouts enclosed in brackets [ ]. The destination **D** is the layout contents of a data record: **Record** "destination" **Layout** that contains the list from [**S**] separated by commas. Example: [Split(11-12), 19, 21, Split(29-32)]

The following script shows an example of creating a list of several roulette layouts to a data record **numbers**.

```
Copy List [Split(11-12), 19, 21, Split(29-32)]
          to Record "numbers" Layout
```

To use the example above, I will create a simple system that demonstrates how to use to use the **Copy List** action command. The roulette layouts, split 11-12, split 28-32, number 19 and number 21 are copied to the layout contents of data record **numbers** to create a list during the system initialization. On every spin, the system will place a 1 unit bet on the list stored in the layout contents of data record **numbers**. This is an easy way to place bets on multiple numbers at the same time.

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
       Copy List [Split(11-12), 19, 21, Split(29-32)]
             to Record "numbers" Layout
    End

    While on Each Spin
    Begin
       Put 1 units on Record "numbers" Layout List
    End
End
```

When you copy information to data records, you can then review them during your system to place bets, change progression amounts, test for certain conditions and so on.

Copy Neighbors

This action command copies the neighboring straight-up numbers from the referenced straight-up number. The quantity of neighboring numbers copied is determine by the **Neighbor Count** action support identifier or by setting the Neighbors field on the main screen of Roulette Xtreme.

For example: I want to copy the neighboring numbers of Number 7. I have already set the neighbor count to 4 and therefore, I expect to copy 4 numbers on each side of Number 7 to the layout contents of a data record. Review the linear subset of the single roulette wheel below.

31, 9, 22, 18, 29, 7, 28, 12, 35, 3, 26

As you can see there are 4 numbers on each side of **7** that will be copied from this action command: 9, 22, 18, 29, 28, 12, 35 and 3. By using the **Put** action command, I can place bets on these number with a single action command. The format is: **Copy Neighbors** **N** **D** where **N** is the referenced straight-up number and **D** the destination. The referenced straight-up number **N** can either be:

- any straight-up roulette number (i.e. **Number 1**, **Number 32**, **Number 0**)
- the layout contents of a data record: **Record** "number" **Layout**. The contents **must only** be a straight-up number.

The destination **D** is the layout contents of a data record: **Record** "destination" **Layout**.

The following script shows an example of creating a list of several roulette layouts to a data record **numbers**.

```
Copy Neighbors of Number 7 to Record "numbers" Layout
```

To use the example above, I will create a simple system that demonstrates how to use to use the **Copy Neighbors** action command. During system initialization, I set the neighbor count to 3 and performed the action command to copy neighboring numbers of **Number 32** to the layout contents of data record **neighbors**. Then on every spin, the system will place a 1 unit bet on the list stored in the layout contents of data record **neighbors**.

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
       Put 3 on Neighbor Count

       Copy Neighbors of Number 32
            to Record "neighbors" Layout
    End

    While on Each Spin
    Begin
       Put 1 units on Record "neighbors" Layout List
    End
End
```

Duplicate

This action command copies the entire contents of a data record to another data record. Both the numeric data and layout contents are copied. This is a great way to temporary store information of a data record. Then at some other time, you can access the temporary data record for some use.

The format is: **Duplicate** **R1** **R2** where **R1** is the entire data record: **Record** **“**record 1**”** and **R2** is the entire destination of a data record: **Record** “record 2”.

The following script shows an example of copying one data record to another.

```
Duplicate Record “record 1” to Record "record 2"
```

To use the example above, I will create a simple system that demonstrates how to use to use the **Duplicate** action command. During system initialization, I create a data record **progression** and set the data contents to 1 as my starting bet. Then I copied the data record **progression** to a temporary record called **temp**. During the system, I checked to see if any number bet has won (in other words, any of the numbers that I placed a bet on) and if so, I would copy the data record **temp** back to the data record **progression** (what this did was reset my progression count) because anytime my numbers lost, I added 1 unit to my progression therefore I needed to reset my progression after a win.

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
       Put 1 on Record "Progression" Data

       Duplicate Record "Progression" to Record "temp"

       Copy Neighbors of Number 21
                to Record "neighbors" Layout

       Put 100% of Record "Progression" Data
                to Record "neighbors" Layout List
       Exit
    End

    If Any Number Bet has Won Each time
    Begin
        Duplicate Record "temp" to Record "Progression"
    End
    Else
    Begin
        Add 1 to Record "Progression" Data
    End

    Copy Neighbors of Number 21 to Record "neighbors" Layout

    Put 100% of Record "Progression" Data
             to Record "neighbors" Layout List
End
```

Miscellaneous Assign action commands

There are other action commands that perform various functions that I will discuss here.

Track Last

I consider this action command most important because it allows you to track the last numbers, even chances, dozens, or any other roulette groups for a certain amount of spins. For example: you can track the last spins (numbers) that have appeared for let say, 37 spins. The results are stored in the layout contents of a data record in a list format like (number 1, number 2, number 3, number 4). From this data record, you can inspect the layout contents and perform some action or check for repeating patterns.

The format is: **Track Last** **S N D** where **S** is the source, **N** is the maximum number to tract and **D** the destination where the last source layout is stored in a list format. The source **S** can either be:

- any valid roulette layout group (i.e. **Number**, **Split**, **Corner**, **Street**, **Line**, **Dozen**, **Column**, **Zero**)
- any valid roulette even chance pairs: (**Even-Odd**, **Odd-Even**, **Black-Red**, **Red-Black**, **High-Low**, **Low-High**)

The maximum number that is used for tracking the last numbers **N** can either be:

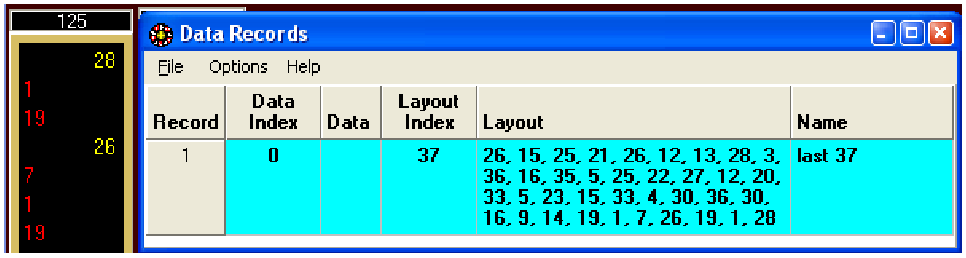
- a whole numeric number (i.e. **1**, **37**, **50**, **100**, not 1.5, 6.75, etc)
- the data contents of a data record: **Record** "number" **Data**, the data contents must be a whole numeric number

The destination **D** is a list of layout contents of a data record: **Record** "destination" **Layout**. (i.e. Red, Red, Black, Red, Black)

The following script shows some examples. The first line will track the last numbers that have appeared within the last 37 spins and place those numbers to the layout contents of data record **last 37** in a form of a list. The second line will track the last even chance pair of color (red/black) for the last 5 spins to the layout contents of a data record **patterns**.

```
Track Last Number for 37 spins to Record "last 37" Layout

Track Last Red-Black for 5 spins Record "patterns" Layout
```

Using the first line example above, the image below shows what happens. As you can see the system will copy the last number to the layout contents of the data record **last 37**. Notice that the entire count is 37 which is the maximum number specified in the command. When all 37 numbers are captured, the 38 and so on number is appended at the end of the list and the top numbers are removed thus always keeping the last 37 numbers available for review and process by other action commands.



Record	Data Index	Data	Layout Index	Layout	Name
1	0		37	26, 15, 25, 21, 26, 12, 13, 28, 3, 36, 16, 35, 5, 25, 22, 27, 12, 20, 33, 5, 23, 15, 33, 4, 30, 36, 30, 16, 9, 14, 19, 1, 7, 26, 19, 1, 28	last 37

Clear

The **Clear** action command does several things.

- It will clear the contents of the **Last Answer** internal variable in the system. The command for this is:
  - **Clear Last Answer**
- It will clear the data and layout contents of an individual data record. The two commands for this are:
  - **Clear Record "**record name**" Data** – removes the numeric data contents and resets the **data index** to 1.
  - **Clear Record "**record name**" Layout** – removes the roulette layout contents and resets the **layout index** to 1.
- It can also clear the data and layout contents of **ALL** data records in your system. This is the quickest way to initialize your system. The command for this is:
  - **Clear All Records** – removes both data and layout contents of all data records and resets their data and layout index counters to 1.
- If you want to clear all data records except for certain ones, then you can do that by specifying **Clear All Records Except [ ]**. The data records that you **DO NOT** want to have its contents cleared are listed between the brackets **[]** enclosed in quotation marks separated by a comma. Example: ["progression", "bankroll balance"]. The system will remove either the data or layout contents of all data records except for those between brackets. The two commands for this are:
  - **Clear All Records Except [**"record 1", "record 2"**] Data** – removes the data contents of **ALL data** records EXCEPT "**name 1**" and "**name 2**" data records.
  - **Clear All Records Except [**"record 1", "record 2" **] Layout** – removes the layout contents of **ALL layout** records EXCEPT "**name 1**" and "**name 2**" data records.

The following script shows some examples of the **Clear** action commands.

```
//clears the internal variable answer
Clear Last Answer
//clears data and layout contents of ALL data records
Clear All Records
//clears only the data contents of data record "bankroll"
Clear Record "bankroll" Data
// clears only the layout contents of data record "last
roulette number"
Clear Record "last roulette number" Layout
//clears only data contents of ALL data records except those
listed between brackets []
Clear All Records Except ["progression", "bankroll"] Data
```

Set Max

The **Set Max** action command sets the data or layout index pointer of a data record to the maximum number of items in its list. If there is only 1 item in the list, then the maximum number is 1, if 2 items in the list, the maximum number is 2 and so on. The two commands are:

- **Set Max Record "**record name**" Data Index** – sets the data index pointer to the number of items in the data contents. If the content is empty, the data index pointer is set to 0.
- **Set Max Record "**record name**" Layout Index** – sets the layout index pointer to the number of items in the layout contents. If the content is empty, the layout index pointer is set to 0.

The following script shows some examples of the **Set Max** action commands.

```
Set Max Record "progression" Data Index

Set Max Record "last roulette number" Layout Index
```

For example: review data contents of the data record **progression** below.

Data contents of data record "progression" = 1,2,4,6,8,16,32

By performing the action command, **Set Max Record "**progression**" Data Index**, the **Data Index** pointer will be set to **7** which is the total number of items in this list. So if I performed the action command of **Add 1** to **Record "**progression**" Data Index**, the data index pointer will be **8**. Then by performing this action command **Put 64** on **Record "**progression**" Data** will append **64** to the end of the list.

A good way to use this action command is when you are trying to append items in a list. I will create a simple system that demonstrates tracking numbers that have repeated twice in a row. When any number repeats 2 times in a row, I set the layout index pointer of data record **tracked numbers** to the maximum items in the list. If initially the list is empty, then the layout index will be set to 0. The next action command **Add** will increment the layout index by 1 thus pointing to the end of the list. Then the following action command **Copy Last** will copy the last number that has appeared (the number that has repeated twice) to the end of the list in the data record. The last action command **Put** will place a 1 unit bet on the layout contents of the numbers that have been added to the data record.

```
System "my system"

Method "main"
Begin
   If Any Number has Hit 2 times
   Begin
      Set Max to Record "tracked numbers" Layout Index
      Add 1 to Record "tracked numbers" Layout Index
      Copy Last Number to Record "tracked numbers" Layout
   End

   Put 1 unit on Record "tracked numbers" Layout List
End
```

When you run this system, the data record **tracked numbers** will contain only numbers that have repeated twice during a session.

Set List

One quick way to create a data list of numbers is to use the **Set List** action command. This popular command is great to create a progression list for betting. (i.e. 5, 10, 20, 40, 80, 160, etc).

The format is: **Set List** [**S**] **D** where [**S**] is the source and **D** the destination. The source [**S**] is a list of numeric data enclosed in brackets [ ]. The destination **D** is the data contents of a data record: **Record** "destination" **Data** that contains the list [**S**] separated by commas. Example: [1, 2, 4, 6, 8, 32, 64]

The following script shows an example of creating a list of several numeric data to a data record **progression**.

```
Set List [1,2,4,6,8,32,64] to Record "progression" Data
```

To use the example above, I will create a simple system that demonstrates how to use to use the **Set List** action command. During system initialization, I will create a betting progression for placing bets on **Lines** or double streets. On every spin, the system will place a progression bet from the data record **progression** to 3 line layouts located in layout contents of the data record **lines**. Prior to placing a bet, the system will check to see if any of the line bets have won and reset the data index pointer to 1 otherwise add 1 to the data index thus pointing to the next number in the list. It will also check to make sure you don't go past the maximum number in the progression list. If so, then reset back to 1 as well.

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
       Set List [1,2,4,6,8,32,64] to
           Record "progression" Data

       Copy List [Line(7-12),Line(16-21),Line(25-30)]
           to Record "lines" Layout
    End

    While on Each Spin
    Begin
       If Any Line Bet has Won Each time
       Begin
           Put 1 to Record "progression" Data Index
       End
       Else
       Begin
          Add 1 to Record "progression" Data Index
          If Record "progression" Data Index > 7
          Begin
             Put 1 to Record "progression" Data Index
          End
       End

       Put 100% of Record "progression" Data to
          Record "lines" Layout List
    End
End
```

Generate Random Number

This action command was added incase someone wanted to experiment with random numbers from a range of **n1** to **n2**. You may want to make a random bet on an even chance layout or use a random number to determine how many spins to track before placing a bet of each session.

The format is: **Generate Random Number** **N1 N2 D** where **N1** is the lowest random number and **N2** is the highest random number. **D** the destination where the random number generated between **N1** and **N2** is stored. So the **N1** and **N2** format is as follows:

- **N1** – lowest random number (i.e. 1, 5, 10, etc). Must be lower than N2
- **N2** – highest random number (i.e. 5, 10, 20). Must be higher than N1

The destination **D** can either be:

- **Bankroll** internal variable, **All Bets** (every layout on the table), **All Outside** (every layout on the outside), **All Inside** (all straight-up numbers), **Record** "record name" **Data** (data contents of a data record)
- Any individual roulette layout (i.e. **Number 19**, **Line (7-12)**, etc).

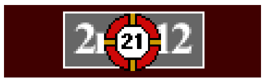
The following script shows an example of generating a random number between 1 and 20 and storing that number into the data contents of data record **wait count**.

```
Generate Random Number from 1 to 20 into Record "wait count"
Data
```

The following system demonstrates how to use the **Generate Random Number** action command. On every spin, the system will generate a random number between 5 and 25 and place that bet to the 2nd dozen.

```
System "my system"

Method "main"
Begin
    While on Each Spin
    Begin
       Generate Random Number from 5 to 25 into 2nd Dozen
    End
End
```

The image below is a sample result from the system above. Note that numeric value of 21 was randomly generated between the numbers 5 and 25.



Assign action command for data flags

The next two action commands pertain to data flags instead of data records.

Set Flag

This action command sets the Boolean value stored in a data flag to either true or false.

The format is: **Set Flag** "flag name" **True**/**False**. Where "flag name" is the unique name of the data flag and **True**/**False** is the Boolean condition. You can only specify **True** or **False** but not both on the same action command line. The following script is an example of setting a data flag called **ready to bet** to False.

```
Set Flag "ready to bet" to False
```

The following system demonstrates how to use the **Set Flag** action command. During system initialization, the data flag **ready to bet** is set to false since we don't want to place any bets at this time. On every spin, the system tests to see if the even chance of black has repeated 5 times in a row and if so, set the data flag **ready to bet** to true to signal the system to place a bet on red hoping the series streak has ended for black.

```
System "my system"

Method "main"
Begin
    While Starting a New Session
    Begin
        Set Flag "ready to bet" to False
    End

    While on Each Spin
    Begin
       If Black has Hit 5 times in a row
       Begin
          Set Flag "ready to bet" to True
       End

       If Flag "ready to bet" is True
       Begin
          Put 5 units on Red
       End
    End
End
```

Reset All Flags

This simple action command will reset the Boolean value stored in all data flags that are in the system to either true or false.

The format is: **Reset All Flags** **True**/**False**. Where **True**/**False** is the Boolean condition. You can only specify **True** or **False** but not both on the same action command line. The following script is an example of setting all data flags to false.

```
Reset All Flags to False
```

The following system demonstrates how to use the **Reset All Flags** action command. Expanding on the previous system that I created from the **Set Flag** section, during system initialization, the data flag **ready to bet** is set to false since we don't want to place any bets at this time. On every spin before testing to see if black has repeated for 5 times in a row, I check to see if the data flag **ready to bet** is set to true and at the same time check to see if the even chance of red has lost a bet 3 times in a row. If both of these conditions are true, then I reset all of the flags to false and start looking for another series streak of black repeating 5 times in a row.

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
        Set Flag “ready to bet” to False
    End

    While on Each Spin
    Begin
       If  Flag “ready to bet” is True
       And Red has Lost 3 times in a row
       Begin
          Reset All Flags to False
       End

       If Black has Hit 5 times in a row
       Begin
          Set Flag “ready to bet” to True
       End

       If Flag “ready to bet” is True
       Begin
          Put 5 units on Red
       End
    End
End
```

Maneuver Action

Maneuver action commands performs various types of maneuvers from moving data through a data record list, searching through history of past occurrences (spins), locating distances on the roulette wheel, causing the program to jump from one section to another (routines) or exit the program all together. These additional action commands help expand the complexity of your system.

Maneuver action commands that alters a list in a data record

The next set of action commands are used to maneuver through a list of a data record either the data section or layout section.

Move List Up

The **Move List Up** action command is used when you need to shift all items in a list of data values or layout values in a data record to the left by some number.

The format is: **Move List Up** **N D** where **N** is the number to cause the items in the list to be shifted to the left by that number and **D** is the destination of the data record that contains the list of items. The destination **D** can either be:

- **Record** "record name" **Data** (i.e. 1, 2, 3, 6, 10, 21)
- **Record** "record name" **Layout** (i.e. 1, 38, 10,20, 00, 18, 32)

As each item is shifted to the left, the first most items will be deleted. You can use this command when you need to keep a certain number of items in the list and you need to append items to the end of the list. The following script shows some examples.

```
Move List Up by 1 on Record "last 5 numbers" Layout

Move List Up by 1 on Record "Sequence" Data
```

The following system demonstrates how to use the **Move List Up** action command. The system will keep track of the last 5 roulette numbers in a data record **spin list**. When the data record **spin list** has more than 5 items in the layout list, the system will use the **Move List Up** and shift the items to the left by 1 and set the **Layout index** to the maximum number of items. The next time a number appears, it will be **appended** to the last position of the data record.

```
System "my system"

Method "main"
Begin
  While on Each Spin
  Begin
     Add 1 to Record "spin list" Layout Index
     Copy Last Number to Record "spin list" Layout

     If Record "spin list" Layout Index > 5 then
     Begin
        Move List Up by 1 of the items located in
           Record "spin list" Layout

        Set Max to Record "spin list" Layout Index
     End
   End
End
```

Move List Down

The **Move List Down** action command is the opposite of the **Move List Up** action command from the previous page. Instead of shifting a list of items in a data record to the left, this command will shift the items in a list to the right.

The format is: **Move List Down** **N D** where **N** is the number to cause the items in the list to be shifted to the right by that number and **D** is the destination of the data record that contains the list of items. The destination **D** can either be:

- **Record** "record name" **Data** (i.e. 1, 2, 3, 6, 10, 21)
- **Record** "record name" **Layout** (i.e. 1, 38, 10,20, 00, 18, 32, Split(17:20))

As each item is shifted to the right, the last most items will be deleted. You can use this command when you need to keep a certain number of items in the list and you need to insert items to the front of the list. The following script shows some examples.

```
Move List Down by 1 on Record "last 5 numbers" Layout

Move List Down by 1 on Record "Sequence" Data
```

The following system demonstrates how to use the **Move List Down** action command. The system will keep track of the last 5 roulette numbers in a data record **spin list**. When the data record **spin list** has more than 5 items in the layout list, instead of appending the items to the list, the system will the **Move List Down** and shift the items to the right by 1 and set the **Layout index** to 0. The next time a number appears, it will be **inserted** to the front (position 1) of the data record.

```
System "my system"

Method "main"
Begin
  While on Each Spin
  Begin
     If Record "spin list" Layout Count > 4 then
     begin
        Move List Down by 1 of the items located in
           Record "spin list" Layout

        Put 0 on Record "spin list" Layout Index
     end

     Add 1 on Record "spin list" Layout Index
     Copy Last Number to Record "spin list" Layout
   End
End
```

To see an actual system that uses both the **Move List Up** and **Move List Down** action commands, visit the Labouchere system written in this book in the **Creating Roulette Systems** chapter.

Locate Number Right

The **Locate Number Right** action command provides a way to locate a number on the roulette wheel that is **n** gaps to the right of the source number. For example on a single zero wheel, if I wanted to know what is 8 gaps to the right of number 32 (the source number), the answer would be number 6 (the final number). The 8 gaps are between number 32 and number 6; therefore, this action command does not include the source number or final number.

The format is: **Locate Number Right** **N S D** where **N** is the number of gaps to count from right of **S** the source number and **D** is the destination data record that will contain the final number. The source **S** can either be:

- the **Last Number** identifier
- **Record** "record name" **Layout** (i.e. 1, 38, 10, 20, 00, 18, 32) – the data record layout **must** only contain roulette straight-up numbers.

The destination **D** is the data record: **Record** "destination" **Layout** that will contain the final roulette number. The following script shows some examples.

```
Locate Number Right 5 gaps of Last Number to
    Record "final number" Layout

Locate Number Right 8 gaps of Record "number" Layout to
    Record "final number" Layout
```

Using the first script above and assuming that number 27 was the last number that has appeared, we can see that number 8 is 5 gaps to the right of number 27 as shown in the single zero layouts below.

```
0, 32, 15, 19, 4, 21, 2, 25, 17, 34, 6, 27, 13, 36, 11, 30, 8, 23, 10, 5,
24, 16, 33, 1, 20, 14, 31, 9, 22, 18, 29, 7, 28, 12, 35, 3, 26
```

The following system demonstrates how to use the **Locate Number Right** action command. On every spin, the system will locate a roulette number that is **8** gaps to the right of the last number that has appeared and store that value to the layout contents of record **number**. Next the system will then copy the last number that has appeared to a data record **last number** and then place a 1 unit bet on both numbers.

```
System "my system"

Method "main"
Begin
  While on Each Spin
  Begin
    Locate Number Right 8 gaps from the Last Number to
        Record "number" layout

    Copy Last Number to Record "last number" Layout

    Put 1 unit on Record "number" Layout
    Put 1 unit on Record "last number" Layout
  End
End
```

Locate Number Left

The **Locate Number Left** action command provides a way to locate a number on the roulette wheel that is **n** gaps to the left of the source number. For example on a single zero wheel, if I wanted to know what is 5 gaps to the left of number 30 (the source number), the answer would be number 6 (the final number). The 5 gaps are between number 30 and number 6; therefore, this action command does not include the source number or final number.

The format is: **Locate Number Left** **N S D** where **N** is the number of gaps to count from left of **S** the source number and **D** is the destination data record that will contain the final number. The source **S** can either be:

- the **Last Number** identifier
- **Record** "record name" **Layout** (i.e. 1, 38, 10, 20, 00, 18, 32) – the data record layout **must** only contain roulette straight-up numbers.

The destination **D** is the data record: **Record** "destination" **Layout** that will contain the final roulette number. The following script shows some examples.

```
Locate Number Left 5 gaps of Last Number to
    Record "final number" Layout

Locate Number Left 8 gaps of Record "number" Layout to
    Record "final number" Layout
```

Using the first script above and assuming that number 6 was the last number that has appeared, we can see that number 21 is 5 gaps to the left of number 6 as shown in the single zero layouts below.

```
0, 32, 15, 19, 4, 21, 2, 25, 17, 34, 6, 27, 13, 36, 11, 30, 8, 23, 10, 5,
24, 16, 33, 1, 20, 14, 31, 9, 22, 18, 29, 7, 28, 12, 35, 3, 26
```

The following system demonstrates how to use the **Locate Number Left** action command. On every spin, the system will locate a roulette number that is **5** gaps to the right of the last number that has appeared and store that value to the layout contents of record **number**. Next the system will then copy the last number that has appeared to a data record **last number** and then place a 1 unit bet on both numbers.

```
System "my system"

Method "main"
Begin
  While on Each Spin
  Begin
    Locate Number Left 5 gaps from the Last Number to
        Record "number" layout

    Copy Last Number to Record "last number" Layout

    Put 1 unit on Record "number" Layout
    Put 1 unit on Record "last number" Layout
  End
End
```

Locate Distance Between

The **Locate Distance Between** action command provides a way to determine the distance between two roulette numbers on the wheel. For example on a single zero wheel, if I wanted to know what is the distance between number 33 and number 29, the answer would be 8 (including number 29).

The format is: **Locate Distance Between** **S1 S2 D** where **S1** is the from-roulette and can either be:

- any roulette straight-up number identifier (i.e. Number 1, Number 10)
- **Record** "record name" **Layout** (i.e. 1, 38, 10, 20, 00, 18, 32) – the data record layout **must** only contain roulette straight-up numbers.

The **S2** is the to-roulette number and can either be:

- any roulette straight-up number identifier (i.e. Number 1, Number 10)
- **Record** "record name" **Layout** (i.e. 1, 38, 10, 20, 00, 18, 32) – the data record layout **must** only contain roulette straight-up numbers.

And **D** contains the distance results stored in the data contents of the data record, **Record** "destination" **Data**. The following script shows some examples.

```
Locate Distance Between Record "from number" Layout &
    Number 19 to Record “gaps” Data

Locate Distance Between Record "from number" Layout &
    Record “to number” to Record “gaps” Data
```

Using the first script above and assuming that number 6 was the last number that has appeared we can see that the distance between number 6 and number 19 is 7 including number 19 as shown in the single zero layouts below. When calculating the distance between two numbers, the system always chooses the shortest distance on the wheel.

```
0, 32, 15, 19, 4, 21, 2, 25, 17, 34, 6, 27, 13, 36, 11, 30,
8, 23, 10, 5, 24, 16, 33, 1, 20, 14, 31, 9, 22, 18, 29, 7,
28, 12, 35, 3, 26
```

The following system demonstrates how to use the **Locate Distance Between** action command. On every spin, the system will find the distance between the last number that has appeared and number 30. If the distance is within 3 spots, a 1 unit bet is placed on number 30 and its neighbors.

```
System “my system”

Method “main"
Begin
  While on Each Spin
  Begin
    Copy Last Number to Record "last number" Layout

    Locate Distance between the Record "last number" Layout
      to Number 30 into Record "distance" Data
```

The rest of this system example is located on the next page.

System continued from previous page

```
        If Record “distance” Data <= 3 then
        Begin
          Copy Neighbors of Number 30 to
             Record “neighbors” Layout
          Put 1 unit on Record “neighbors” Layout List
      End
    End
```

Locate Number Backward

The **Locate Number Backward** action command provides a cool way to find previous roulette numbers that have appeared **n** spins (occurrences) ago. This is kind of looking back into history. For example: on a double zero wheel note the spin occurrence listed below. Number 1 is the last number that has appeared. If I wanted to know what roulette number appeared 11 spins ago, the answer would be number 20 since it is 11 spins from the last spin.

```
00, 27, 10, 25, 29, 12, 8, 19, 31, 18, 6, 21, 33, 16, 4, 23,
35, 14, 2, 0, 28, 9, 26, 30, 11, 7, 20, 32, 17, 5, 22, 34,
15, 3, 24, 36, 13, 1 (last spin)
```

You can also use this action command to find a roulette number that has appeared based on another roulette number that had previous appeared. For example, using the same spin occurrence from above, I wanted to know what roulette number appeared 6 spins ago from the last time number 9 appeared. Looking at the list, I see that number 23 appeared 6 spins before number 9.

The format is: **Locate Number Backward** **N S D** where **N** is the number of spins to look backwards. **N** can either be:

- An integer (whole) number (i.e. 1, 10, 20, 15, 4)
- **Record** "record name" **Data** (i.e. 5, 10, 1, 100) – contains the data value of any integer whole number.

The **S** is the source identifier that is the starting point of the search. The source **S** can either be:

- the **Last Number** identifier (last number that has appeared)
- **Record** "record name" **Layout** (i.e. 1, 38, 10, 20, 00, 18, 32) – the data record layout **must** only contain roulette straight-up numbers.

The destination **D** is the data record: **Record** "destination" **Layout** that will contain the final roulette number that was found **N** spins ago from **S**. The following script shows some examples.

```
Locate Number Backward 10 spins of Last Number to
    Record "found number" Layout

Locate Number Backward 8 spins of Record "number" Layout to
    Record "found number" Layout

Locate Number Backward Record "# of spins" Data from
    Last Number to Record "found number" Layout
```

The following system demonstrates how to use the **Locate Number Backward** action command. On every spin, the system will generate a number from 1 to 20 and store that number to data record **spins**. Once the random number has been generated, it will be used to locate a roulette number that has appeared by the random number before the last number. So, basically, the found number could be anywhere from 1 to 20 spins backwards from the last number that has appeared. Once found, the system then will place a 5 unit bet the dozen layout of the found roulette number. This should be interesting. The system is located on the next page.

The system using **Locate Number Backward** action command.

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
        Put 5 units on Record "bet" Data
    End

    While on Each Spin
    Begin
        Generate Random Number from 1 to 20
           into Record "spins" Data

        Locate Number Backward Record "spins" Data spins
           from the Last Number to Record "number" Layout

        Put 100% of Record "bet" Data to
          Dozen Nearest of Record "number" Layout
    End
End
```

Locate Number Forward

The **Locate Number Forward** action command is similar to **Locate Number Backward** except you are searching for numbers that have appeared forward in time from a roulette number that had appeared sometime ago and you wanted to know if some roulette number appeared **n** spins in front of that number. For example: on a double zero wheel note the spin occurrence listed below. Number 1 is the last number that has appeared. If I wanted to know what roulette number appeared 9 spins after number 28 in the list below, the answer would be number 5 since it appeared 9 spins from number 28.

```
00, 27, 10, 25, 29, 12, 8, 19, 31, 18, 6, 21, 33, 16, 4, 23,
35, 14, 2, 0, 28, 9, 26, 30, 11, 7, 20, 32, 17, 5, 22, 34,
15, 3, 24, 36, 13, 1 (last spin)
```

You can also use this action command to find a roulette number that has appeared n spins from the beginning of your session by using the identifier **Beginning Session**. For example, using the same spin occurrence from above, I wanted to know what roulette number appeared 32 spins from the beginning session. The first number that had appeared when the session was started in the number 00, so 32 spins later the number is number 15.

The format is: **Locate Number Forward** **N S D** where **N** is the number of spins to look forward. **N** can either be:

- An integer (whole) number (i.e. 1, 10, 20, 15, 4)
- **Record** "record name" **Data** (i.e. 5, 10, 1, 100) – contains the data value of any integer whole number.

The **S** is the source identifier that is the starting point of the search. The source **S** can either be:

- the **Beginning Session** identifier (first number that appeared when the session started)
- **Record** "record name" **Layout** (i.e. 1, 38, 10, 20, 00, 18, 32) – the data record layout **must** only contain roulette straight-up numbers.

The destination **D** is the data record: **Record** "destination" **Layout** that will contain the final roulette number that was found **N** spins forward from **S**. The following script shows some examples.

```
Locate Number Forward 10 spins after Beginning Session to
    Record "found number" Layout

Locate Number Forward 8 spins after Record “number” Layout
    to Record "found number" Layout

Locate Number Forward Record “# of spins” Data after
    Beginning Session to Record “found number” Layout
```

The following system demonstrates how to use the **Locate Number Forward** action command. On every spin, the system will generate a number from 1 to 80 and store that number to data record **spins**. Once the random number has been generated, it will be used to locate a roulette number that has appeared by the random number from the beginning session. So, basically, the found number could be anywhere from 1 to 80 spins forward from the start of the session. Once found, the system then will place a 5 unit bet the dozen layout of the found roulette number. The system is located on the next page.

The system using **Locate Number Forward** action command.

```
System “my system”

Method “main"
Begin
    While Starting a New Session
    Begin
        Put 5 units on Record "bet" Data
    End

    While on Each Spin
    Begin
        Generate Random Number from 1 to 80
           into Record "spins" Data

        Locate Number Forward Record "spins" Data spins
           from the Beginning Session to Record "number"
           Layout

        Put 100% of Record "bet" Data to
          Dozen Nearest of Record "number" Layout
    End
End
```

Call

The **Call** action command is a very important command. It allows your system to jump to a method (sub-routine) based on its name enclosed in quotes. Below is an example of a call action command.

```
Call “make bets”
```

When the program see this command, the program will branch (jump) to the method **make bets**. There the program will process all commands within that method and then return back to this line and start processing other commands going forward. Calling methods allows you to process the same routines over and over within your system without having to duplicate them. Here is a simple system that is using the **Call** action command to jump to method **place bets** several times without duplication.

```
System “my system”

Method “main"
Begin
     If Black has Hit for 3 times
     Begin
        Call “place bets”
     End

     If Number 0 has Hit Each time
     Begin
        Call “place bets”
     End
End

Method “place bets”
Begin
   Put 1 unit on List [13, 21, 32]
End
```

Return

Normally, when you enter a method, the system will process all commands until it finds the **End** identifier. At that point the program is returned back to it calling action statement. However, sometimes it may be necessary to return from your method earlier based on some condition. To do this you would use the **Return** action command.

On the next page, I will show you the same system as above, but instead of the system performing all commands inside the method, I added a condition command in the method **place bets** to check if any even number has appeared. If it has then return and do not place any bets on the numbers 13, 21 and 32. The **Return** action command causes the program to return back to the calling program immediately.

The simple system is located on the next page.

Simple system using the **Return** action command inside a method.

```
System “my system”

Method “main"
Begin
     If Black has Hit for 3 times
     Begin
        Call “place bets”
     End

     If Number 0 has Hit Each time
     Begin
        Call “place bets”
     End
End

Method “place bets”
Begin
   If Even has hit each time
   Begin
     //return back to its calling action statement
      Return
    End

   Put 1 unit on List [13, 21, 32]
End
```

Exit

The **Exit** action command is similar to the **Return** action command except instead of the program returning back to its calling action command; the system will exit to the main screen and wait for another input either from the keyboard or auto-processing spins. Here is the simple system using the **Exit** action command inside a method.

```
System “my system”

Method “main"
Begin
     If Black has Hit for 3 times
     Begin
        Call “place bets”
     End

     If Number 0 has Hit Each time
     Begin
        Call “place bets”
     End
End

Method “place bets”
Begin
   If Even has hit each time  Begin
      Exit  //go back to the main screen now
    End

   Put 1 unit on List [13, 21, 32]
End
```

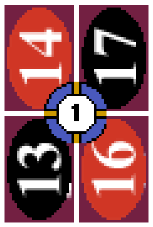
Working With Roulette Layouts

4

There are so many ways to place bets on a roulette board that I figure I need to discuss the different options and their proper syntax when creating a system using RX Scripting. Now, I assume you already know how to play roulette and what the payouts are for each layout so I won't go into those details. If you don't know, there are many books on roulette that discuss the different combinations and payouts.

Remember, the beauty of RX Scripting is the almost English like sentence when designing a system. Therefore, when referencing a roulette layout, you refer to it by its name (i.e. **Number 1**, **Number 19**, and so on).

However, there are some layouts that are complex and the way you refer to them when playing a roulette game may be different than creating a system. For example: I want to place a corner bet of 1 unit on 13, 14, 16, and 17 as shown in image example below.



To write that in RX Scripting as a system, you would do the following:

```
System "my system"

Method "main"
Begin
   Put 1 unit on Corner(13-17)
End
```

If you know how to place corner bets on an actual roulette game, then clearly the above syntax for the corner bet is easy to understand. You also use the same format when writing a condition statement as well as noted in the next script example.

```
System "my system"
// if corner 13,14,16,17 has won a bet, add 1 unit
Method "main"
Begin
   If Corner(13-17) has Won Each time
   Begin
      Add 1 unit to Corner(13-17)
   End
End
```

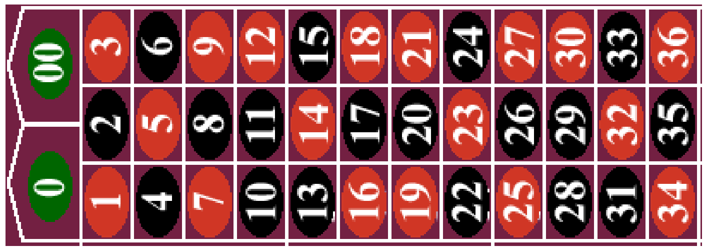
In the next few pages, I'll go over the different roulette layouts and their proper syntax when creating a system with RX Scripting.

Inside Layouts

Inside layouts refer to all layouts where bets can be placed on the field of numbers in the center portion of the roulette table.

Individual Straight-up Numbers

As shown in the table below, the individual numbers are represented by the following syntax **Number 21** which refers to the number 21 on the roulette table. Likewise, **Number 00** represents the double zero on the American roulette table.

Roulette layout	Their meaning
Number 1 … Number 36	The individual number on the table from 1 to 36.
Number 0	The individual single zero on the table
Number 00	The individual double zero on the table
	

When referencing the individual numbers, you always prefix the number by the actual word **Number**. Example, to place a bet on number 36, enter the following:

```
System "my system"

Method "main"
Begin
   Put 1 unit on Number 36
End
```

There are two exceptions to this. When referencing the list identifier with either the action command of **List** and **Copy List**, or the condition command **List**. Between the brackets, you can omit the word **Number** as this is optional. Script below shows the alternative method for these two commands.

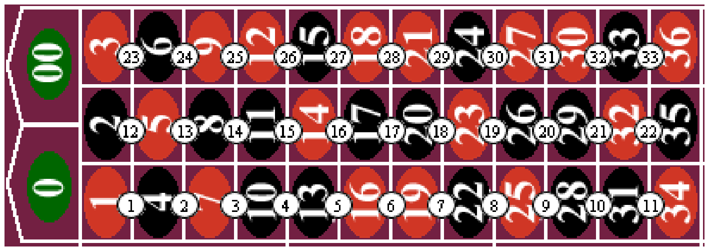
```
System "my system"

Method "main"
Begin
   When Starting a New Session
   Begin
      Copy List [21, 13, 19, 32, 00, 18] to
          Record "numbers" Layout
   End

   If List [21, 00, 6, 36] have Not Hit Each time
   Begin
      Put 1 units on Record "numbers" Layout List
      Put 2 units on List [21, 00, 6, 36]
   End
End
```

Split

Split layouts are used when referencing any number pairs that are adjacent to each other on the roulette table. Since there are many split combinations, this book divides the split layouts into two categories: **horizontal split** and **vertical split**

Horizontal Split

The horizontal split layouts are numbers that are side by side of each other as noted on the roulette picture from the Roulette Xtreme software. The table below shows all the possible horizontal roulette split layouts and their corresponding location # on the roulette board.

Horizontal Split layout	#	**Horizontal Split layout**	#	**Horizontal Split layout**	#
Split(1-4)	1	Split(2-5)	12	Split(3-6)	23
Split(4-7)	2	Split(5-8)	13	Split(6-9)	24
Split(7-10)	3	Split(8-11)	14	Split(9-12)	25
Split(10-13)	4	Split(11-14)	15	Split(12-15)	26
Split(13-16)	5	Split(14-17)	16	Split(15-18)	27
Split(16-19)	6	Split(17-20)	17	Split(18-21)	28
Split(19-22)	7	Split(20-23)	18	Split(21-24)	29
Split(22-25)	8	Split(23-26)	19	Split(24-27)	30
Split(25-28)	9	Split(26-29)	20	Split(27-30)	31
Split(28-31)	10	Split(29-32)	21	Split(30-33)	32
Split(31-34)	11	Split(32-35)	22	Split(33-36)	33
					

When referencing the horizontal split layout, you type the word **Split** followed by in parenthesis () the 2 numbers that are back-to-back together with a hyphen **-** in between the numbers, (i.e. 10-13). For example to place a bet on split 17,20, enter the following:

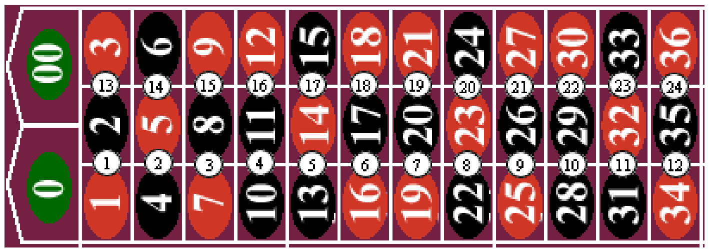
```
System "my system"

Method "main"
Begin
   Put 1 unit on Split(17-20)
End
```

As noted in the table above, split 17, 20 is represented by the # 17 on the roulette table image.

Vertical Split

The vertical split layouts are numbers that are stack on top of each other as noted on the roulette picture from the Roulette Xtreme software. The table below shows all the possible vertical split layouts and their corresponding location # on the roulette board.

Vertical Split layout	#	Vertical Split layout	#
Split(1-2)	1	Split(2-3)	13
Split(4-5)	2	Split(5-6)	14
Split(7-8)	3	Split(8-9)	15
Split(10-11)	4	Split(11-12)	16
Split(13-14)	5	Split(14-15)	17
Split(16-17)	6	Split(17-18)	18
Split(19-20)	7	Split(20-21)	19
Split(22-23)	8	Split(23-24)	20
Split(25-26)	9	Split(26-27)	21
Split(28-29)	10	Split(29-30)	22
Split(31-32)	11	Split(32-33)	23
Split(34-35)	12	Split(35-36)	24



When referencing the vertical split layout, you type the word **Split** followed by in parenthesis () the 2 numbers that are back-to-back together with a hyphen **-** in between the numbers, (i.e. 16-17). For example to place a bet on split 26,27, enter the following:

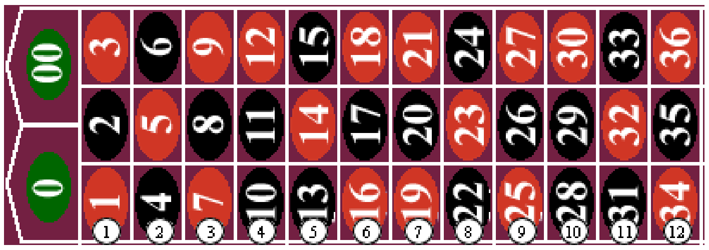
```
System “my system”

Method “main"
Begin
   Put 1 unit on Split(26-27)
End
```

As noted in the table above, split 26, 27 is represented by the # 21 on the roulette table image.

Street

Street layouts are used when referencing any three-number combination on any of the three-number rows on the roulette table. The table below shows all the possible street layouts and their corresponding location # on the roulette board.

Street layout	**#**	**Street layout**	**#**	**Street layout**	**#**
Street(1-3)	1	Street(13-15)	5	Street(25-27)	9
Street(4-6)	2	Street(16-18)	6	Street(28-30)	10
Street(7-9)	3	Street(19-21)	7	Street(31-33)	11
Street(10-12)	4	Street(22-24)	8	Street(34-36)	12



When referencing the street layout, you type the word **Street** followed by in parenthesis () the 2 numbers that are the lowest and highest number of the three-number row together with a hyphen **-** in between the numbers, (i.e. 1-3). For example to place a bet on street 10,11,12, enter the following:

```
System “my system”

Method “main"
Begin
   Put 1 unit on Street(10-12)
End
```

As noted in the above table, street 10, 11, 12 is represented by the # 4 on the roulette table image.

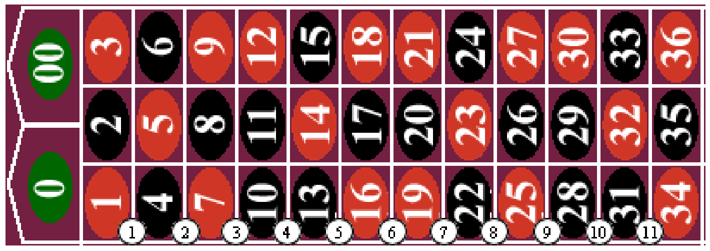
Below is an example of checking if street 19, 20, 21 has appeared each time and if so, place bets on this street and both sides of the street.

```
System “my system”

Method “main"
Begin
   If Street(19-21) has Hit Each time
   Begin
      Put 1 unit on List [16, 17, 18, 22, 23, 24]
      Put 1 unit on Street(19-21)
   End
End
```

Line (Double Street)

Line layouts are used when referencing any six-number combination on two adjacent rows of the roulette table. This is also known as the Double Street. The table below shows all the possible line layouts and their corresponding location # on the roulette board.

Line layout	#	Line layout	#
Line(1-6)	1	Line(19-24)	7
Line(4-9)	2	Line(22-27)	8
Line(7-12)	3	Line(25-30)	9
Line(10-15)	4	Line(28-33)	10
Line(13-18)	5	Line(31-38)	11
Line(16-21)	6		



When referencing the line layout, you type the word **Line** followed by in parenthesis () the 2 numbers that are the lowest and highest number of the six-number combination together with a hyphen - in between the numbers, (i.e. 1:6). For example to place a bet on line 19,20,21,22,23,24, enter the following:

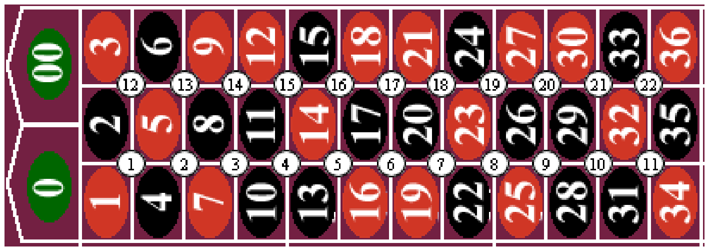
```
System "my system"

Method "main"
Begin
   Put 1 unit on Line(19-24)
End
```

As noted in the above table, line 19, 20, 21, 22, 23, 24 is represented by the # 7 on the roulette table image. The next script example shows how to check to see if any line bet has won and if so, reset to 1 unit otherwise add 1 unit to three lines of 7-12, 16-21, and 25-30.

```
System "my system"

Method "main"
Begin
    If Any Line Bet won each
    Begin
        Put 1 on List [Line(7-12), Line(16-21), Line(25-30)]
    End
    Else
    Begin
        Add 1 to List [Line(7-12), Line(16-21), Line(25-30)]
    End
End
```

Corner

Corner layouts are used when referencing any four-number combination on any square block of four numbers on the roulette table. The table below shows all the possible corner layouts and their corresponding location # on the roulette board.

Corner layout	**#**	**Corner layout**	**#**
Corner(1-5)	1	Corner(2-6)	12
Corner(4-8)	2	Corner(5-9)	13
Corner(7-11)	3	Corner(8-12)	14
Corner(10-14)	4	Corner(11-15)	15
Corner(13-17)	5	Corner(14-18)	16
Corner(16-20)	6	Corner(17-21)	17
Corner(19-23)	7	Corner(20-24)	18
Corner(22-26)	8	Corner(23-27)	19
Corner(25-29)	9	Corner(26-30)	20
Corner(28-32)	10	Corner(29-33)	21
Corner(31-35)	11	Corner(32-36)	22
			

When referencing the corner layout, you type the word **Corner** followed by in parenthesis () the 2 numbers that are the lowest and highest number of the four-number square together with a colon : in between the numbers, (i.e. 1:5). For example to place a bet on corner 14,15,17,18, enter the following:

```
System “my system”

Method “main"
Begin
   Put 1 unit on Corner(14-18)
End
```

As noted in the previous table, corner 14,15,17,18 is represented by the # 16 on the roulette table image.

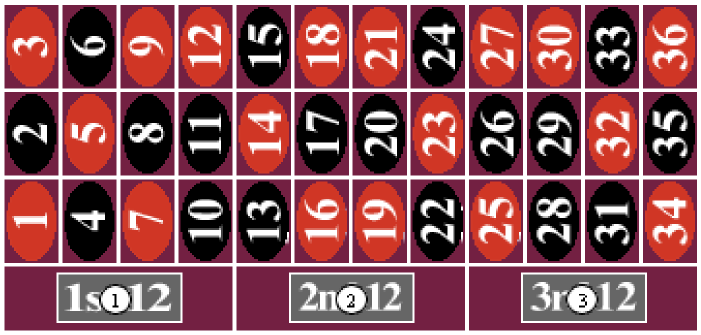
Outside Layouts

Outside layouts refers to all layouts where bets can be placed that are outside of the main field of numbers of the roulette table.

Dozen

Dozen layouts are outside layouts that reference a set of twelve numbers on the roulette table. The table below shows all the possible dozen layouts and their corresponding location # on the roulette board.

Dozen layout	Set of 12 Roulette Numbers	#
1st Dozen	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12	1
2nd Dozen	13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24	2
3rd Dozen	25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36	3



On the Roulette Xtreme image from table above, the **1st 12** section represents numbers from 1 to 12, the **2nd 12** section represents numbers from 13 to 24 and the **3rd 12** section represents numbers from 25 to 36. When referencing any one of the dozen layouts with RX Scripting, you type the dozen # such as **1st**, **2nd**, or **3rd** followed by the word **Dozen**. For example to place a bet on 2nd dozen which are numbers from 13 to 24, enter the following:

```
System “my system”

Method “main"
Begin
   Put 1 unit on 2nd Dozen
End
```

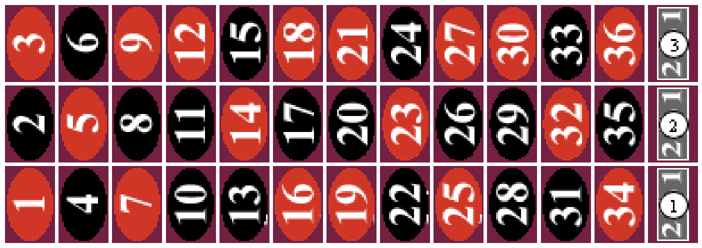
As noted in above table, 2nd dozen of numbers 13 to 24 is indicated by the #2 on the roulette table image. Below is an example of checking if the 2nd dozen has appeared each time and if so, place bets on the other two dozens.

```
System “my system”

Method “main"
Begin
   If 2nd Dozen has Hit Each time
   Begin
      Put 5 units on List [1st Dozen, 3rd Dozen]
   End
End
```

Column

Column layout identifiers are outside layouts that reference a set of twelve column numbers spanning from left to right on the roulette table. The table below shows all the possible column layouts and their corresponding location # on the roulette board.

Column layout	Set of 12 Roulette Numbers	#
Column A	1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34	1
Column B	2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35	2
Column C	3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36	3



On the Roulette Xtreme image from the above, the numbers 1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34 represents column A, the numbers 2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35 represents column B and the numbers 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36 represents column C. When referencing any one of the column layouts with RX Scripting, you type the word **Column** followed by the letter **A**, **B**, or **C**.

For example to place a bet on Column C which are numbers 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, enter the following:

```
System “my system”

Method “main"
Begin
   Put 1 unit on Column C
End
```

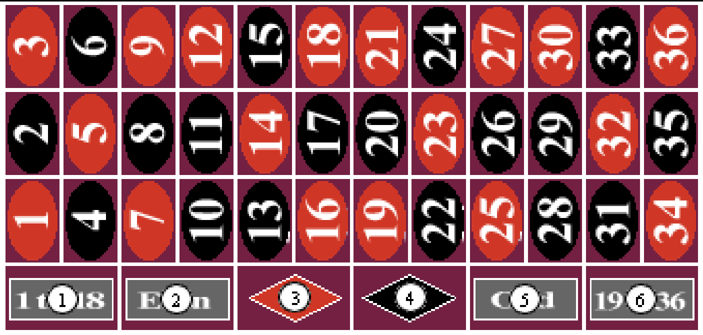
As noted in above table, **Column C** is indicated by the # 3 on the roulette table image.

Below is an example of checking if the column B has appeared each time and if so, place bets on the other two columns and to the 1st and 2nd dozens.

```
System “my system”

Method “main"
Begin
   If 2nd Dozen has Hit Each time
   Begin
      Put 5 units on List [1st Dozen, 2nd Dozen]
      Put 5 units on List [Column A, Column B]
   End
End
```

Even Chance

There are 6 even chance bets that consists of 18 roulette numbers group together. The table below shows all the possible even chance layouts and their corresponding location # on the roulette board. When placing a bet on an even change layout, you place it on one of the #s listed in table below of the roulette image.

Even Chance layout	Set of 18 Roulette Numbers	#
Low	1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18	1
Even	2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32, 34,36	2
Red	1,3,5,7,9,12,14,16,18,19,21,23,25,27,30,32,34, 36	3
Black	2,4,6,8,10,11,13,15,17,20,22,24,26,29,28,31, 33,35	4
Odd	1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33, 35	5
High	19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,3 4,35,36	6
		

To place a bet on all low numbers which are from 1 to 18, you place your unit on the section marked 1 to 18. When creating a system with RX Scripting, you type the word **Low** which tells the system that you are referring to the low numbers of 1 to 18 as shown in the script below.

```
System “my system”

Method “main"
Begin
   Put 1 unit on Low
End
```

The same is true for the high numbers of 19 to 36. The RX Scripting identifier word is **High** which tells the system that you are referring to the high numbers of 19 to 36.

```
System “my system”

Method “main"
Begin
   Put 1 unit on High
End
```

Using Layouts with Commands

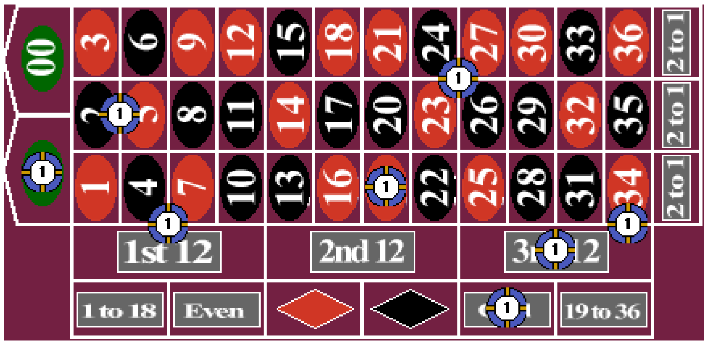
Now that I have discussed the various roulette layouts and their proper syntax with RX Scripting, I will give some examples of their use with action and condition commands.

Action Command

I will write a script using only the action commands to place a 1 unit bet on the following layouts: number 19, single 0, 3rd dozen, even chance odd, corner 23,24,26 and 27, line (or double street) 4 to 9, street 34 to 36 and split 2,5. The script below shows the proper way to write the roulette layouts.

```
System "my system"
{
to make a 1 unit bet of certain layouts
{
Method "main"
Begin
   Put 1 unit on Number 19
   Put 1 unit on Number 0
   Put 1 unit on 3rd Dozen
   Put 1 unit on Odd
   Put 1 unit on Corner(23:27)
   Put 1 unit on Line(4-9)
   Put 1 unit on Street(34-36)
   Put 1 unit on Split(2-5)
End
```

The results are shown on the roulette image below.



The next script shows an alternate way to place bets on the same roulette layouts as noted on the previous image by using the **List** identifier. Note the omission of the layout identifier **Number** from the 19 and 0 as it is optional.

```
System "my system"
{
to make a 1 unit bet of certain layouts
}
Method "main"
Begin
   Put 1 unit on List [19, 0, 3rd Dozen, Odd,
      Corner(23:27), Line(4-9), Street(34-36),
      Split(2-5)]
End
```

Condition Command

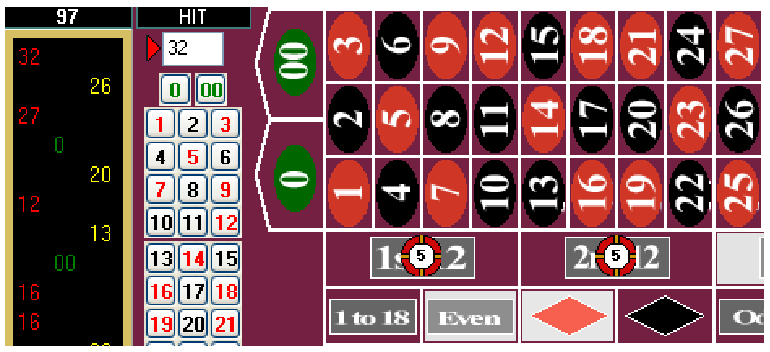
When writing a condition command one must think of what type of event to look for before performing some action command. To do this, I must jot down what I want to do before transferring it to RX Scripting. The system I am creating is based on this example:

1. When any dozen has repeated 3 times in a row
2. Place a 5 unit bet on the other two dozens

I have my system jotted down on paper or in my thoughts, so I will create the system using RX Scripting as shown in the following script.

```
System “my system”
{
When a dozen has repeated 3 times in a row, place a bet on
the other two dozens
}
Method “main”
Begin
   If 1st Dozen has Hit 3 times in a row
   Begin
     Put 5 units on List [2nd Dozen, 3rd Dozen]
   End

   If 2nd Dozen has Hit 3 times in a row
   Begin
     Put 5 units on List [1st Dozen, 3rd Dozen]
   End

   If 3rd Dozen has Hit 3 times in a row
   Begin
     Put 5 units on List [1st Dozen, 2nd Dozen]
   End

End
```

The system consists of 3 different condition events. First, I checked to see if the 1st dozen has appeared for 3 times in row and if this happened, place a 5 unit bet on the other two dozens. The same format is used for the other two dozens as well. From the marquee board on the image below, the 3rd Dozen (numbers 27, 26 and 32) have appeared for 3 times. This caused the condition statement to evaluate as true and the two action statements were executed which placed a 5 unit bet on the 1st and 2nd dozens.



CREATING ROULETTE SYSTEMS

5

Ok, now you have a refresher course on how to use the Action and Condition commands with roulette layouts and data records. It is time to create some system from the very basic to the very complex. I will provide step-by-step instructions so it will be easy for you to follow and reproduce.

Martingale System

The Martingale system is simply doubling your bet after each loss. This is used mostly playing on the even chance layouts (red, black, odd, even, high and low) and it is by far the most popular betting system in the world. Let's take an example of how this type of betting system works.

BET #	UNITS BET	RESULT	NET GAIN
1	1	lose	-1
2	2	lose	-3
3	4	lose	-7
4	8	lose	-15
5	16	win	+1

No matter what step you are when you win, you will always make a profit of 1 unit. As you can see, it is achievable to make a profit of 1 unit by performing the doubling of bets after each loss, however, one would need a very large bankroll to sustain such a bad run and most casinos have table limits that prevent you from making such large bets. Some Martingale systems have been modified to use some sort of progression list instead of just the normal doubling up process. This will allow you to adjust your maximum bankroll risk and also allow you to extend the possible bad streak far out as possible. Here are some examples of a martingale progression lists:

- **1,2,4,8,16,32,64,128,256,512**
  normal list that covers 10 losses in a row, high bankroll needed
- **0,0,0,0,1,1,3,6,12,24,48,96,192**
  modified list that covers 13 loses in a row, using hypothetical and insurance bets
- **0,0,0,0,1,1,3,6,12,24,48**
  modified list that covers 11 loses in a row, same as previous type but good for low bankroll players.
- **1,2,4,2,2,4,2,3,5,2,3,6,2,4,6**
  modified list that covers 15 loses in a row, uses insurance bets and reduce profits results

The Martingale system that is created in this book will use a modified progression list which uses a combination of insurance bets and reduce profits. This will allow losing 15 times in a row before losing all of your bankroll. The total bankroll for this system is 48 units. The system will place bets on the even chance layout of Red and the algorithm is written like this:

<u>When a new session is started, do this…</u>

- Set the **progression** list to a data record
- Place the first initial bet on Red

<u>On every spin, check for…</u>

- On a win, reset the progression list to first step
- On a loss, increase to the next progression level
  - Check progression level to determine if reached maximum level. If maximum level reached, then **stop session**, otherwise
- Place another bet on Red at current **progression** level.

Martingale system written in RX Scripting from the algorithm above.

```
System “Martingale”
{
Uses a progression betting list for even chance Red
}
Method “main”
Begin
   While Starting a New Session
   Begin
     Set List [1,2,4,2,2,4,2,3,5,2,3,6,2,4,6] to
         Record "progression" Data

     Put 100% of Record "progression" Data to Red
   End

   While on Each Spin
   Begin
      If Any Even Bet has Won Each time
      Begin
         Put 1 on Record "progression" Data Index
      End

      If Any Even Bet has Lost Each time
      Begin
         Add 1 to Record "progression" Data Index

         If Record "progression" Data Index >
              Record "progression" Data Count
         Begin
            Stop Session
         End
      End

      Put 100% of Record "progression" Data to Red
    End
End
```

With this system, you can have two options.

The first option is you can change and set your own progression list between the brackets of the **Set List** action command to suite you maximum bankroll risk. The items in the list can be of any quantity. For example, you can create a list like: [1,2,4,8,16,32] which is considered aggressive and requires a bankroll of 63 unit or like: [1,1,1,1,2,4,6,8,16] which is considered a low risk and requires a bankroll of 32 units. There are lots of combinations you can use for this betting system.

The second option is you can change the even chance of Red to any other even chance layout of your choosing. All of this is done within the RX Scripting prior to running a session.

Expanding on the Martingale System

Suppose you want to be able to select what even chance layout prior to placing any bets. The solution to this is use a dialog screen and let the user select the layout type. The previous system will be expanded to include the ability to have the user select a layout type using an **Input Dropdown** action command.

The expanded system will place bets on the even chance layout (user choice) and the algorithm is written like this (the changes are in *Italic*):

When a new session is started, do this…

- Set the **progression** list to a data record
- *Call Input routine to allow user to select layout type*
- Place the first initial bet on *data record* ***even chance***

On every spin, check for…

- On a win, reset the **progression** list to first step
- On a loss, increase to the next progression level
    - Check progression level to determine if reached maximum level. If maximum level reached, then **stop session**, otherwise
- Place another bet on *data record* ***even chance*** at current **progression** level

Method "Input" routine here…

- *Input "Select even chance layout type of: (Red, Black, Odd, Even, High, Low)*
- *Copy this layout type to a data record called* ***even chance***

As you can see from the algorithm above, the changes are quite minor and a new method was created to handle the input routine. The entire system written in RX Scripting is shown below and continued on the next page.

```
System "Modified Martingale"
{
Uses a progression betting list for even chance choices
}
Method "main"
Begin
   While Starting a New Session
   Begin
     Set List [1,2,4,2,2,4,2,3,5,2,3,6,2,4,6] to
         Record "progression" Data

      Call "Input"

     Put 100% of Record "progression" Data to
       Record "even chance" Layout
   End

   While on Each Spin
   Begin
      If Any Even Bet has Won Each time
      Begin
```

System continued from previous page

```
            Put 1 on Record "progression" Data Index
         End

         If Any Even Bet has Lost Each time
         Begin
            Add 1 to Record "progression" Data Index

            If Record "progression" Data Index >
                 Record "progression" Data Count
            Begin
               Stop Session
            End
         End

         Put 100% of Record "progression" Data to
             Record “even chance” Layout
       End
   End

   Method "Input"
   Begin
       Input Dropdown "Select even chance layout type
               1:=Red
               2:=Black
               3:=Odd
               4:=Even
               5:=High
               6:=Low" to Record "type" Data

       If Record "type" Data = 1 then Begin
          Copy Red to Record "even chance" Layout   End
       If Record "type" Data = 2 then Begin
          Copy Black to Record "even chance" Layout End
       If Record "type" Data = 3 then Begin
          Copy Odd to Record "even chance" Layout   End
       If Record "type" Data = 4 then Begin
          Copy Even to Record "even chance" Layout  End
       If Record "type" Data = 5 then Begin
          Copy High to Record "even chance" Layout  End
       If Record "type" Data = 6 then Begin
          Copy Low to Record "even chance" Layout   End
   End
```

Within the method **Input**, the **Input Dropdown** action command can only stored numeric data which is denoted by the ***n***:= *choice name* (i.e. 1:=Red). The ***n*** is the numeric number that is store into a data record. So, after the user selects which even chance layout from the dropdown list of choices, the system then needs to perform a **If**...then logic condition command to determine which numeric number has been stored. Once the logic of the condition command becomes true, the system then copies the appropriate even chance layout to the data record **even chance**. The stored layout in the data record is then used in the main method section to place bets.

D'Alembert System

The D'Alembert system is a simple betting progression system in where after each loss, one unit is added to the next bet, and after each win, one unit is deducted from the next bet. Usually you start with approximately 5 or 10 times the minimum bet to all for 5 to 10 wins in a row so you don't hit you minimum table limit too soon. Like the Martingale system, this is used mostly playing on the even chance layouts (red, black, odd, even, high and low). Let's take an example of how this type of betting system works.

BET #	UNITS BET	RESULT	NET GAIN
1	5	lose	-5
2	6	win	+1
3	5	lose	-4
4	6	win	+2
5	5	win	+7
6	4	lose	+3

As you can see from the example above, there are 3 wins and 3 losses. The mathematical formula for this is whenever the number of wins equals the number of losses; the net gain is equal to the number of wins. However, due to the outcome of the 0 and or 00 (for American wheels), you can average about 18 wins and 20 losses every 38 spins. This type of system is best used on a single 0 wheel with Le Partage or En Prison rule. The D'Alembert system that is created in this book will use the **follow the color** option of red and black. In addition, the system will ask the user for a starting bet amount but will default to. The algorithm is written like this:

When a new session is started, do this…

- Set the initial betting amount of 15 to a data record **amount**
- Set the initial table minimum limit of 1 to data record **minimum**
- Call **Input** routine to allow user to select an initial amount to bet and table minimum bet

On every spin, check for…

- On a win, decrease bet by 1 by subtracting 1 from data record **amount**
    - Check if minimum table bet has been reached and if so, stop session.
- On a loss, increase bet by 1 unit by adding 1 to data record **amount**
    - Check if **Bankroll** has been depleted and if so, stop session
- Copy the last color that has appeared to data record **color**
- Place a bet on data record **color** at using the data record **amount**
- If 0 appears, bet on last color

Method "Input" routine here…

- Input "Enter starting initial bet" (default is 15)
- Input "Enter minimum table bet" (default is 1)

The entire system written in RX Scripting is shown on the next page.

D’Alembert system written in RX Scripting from the algorithm above.

```
System "D’Alembert"
{
Uses the D’Alembert progression system for the "follow the
color" session
}
Method "main"
Begin
   While Starting a New Session
   Begin
     Put 15 on Record "amount" Data
     Put 1  on Record "minimum" Data

     Call "Input"
   End

   While on Each Spin
   Begin
      If Any Even Bet has Won Each time
      Begin
         Subtract 1 on Record "amount" Data

         If Record "amount" Data < Record "minimum" Data
         Begin
            Stop Session
         End
      End

      If Any Even Bet has Lost Each time
      Begin
         Add 1 to Record "amount" Data

         If Bankroll <= 0
         Begin
            Stop Session
         End
      End

      Copy Last Red-Black to Record "color" Layout

      Put 100% of Record "amount" Data
            to Record "color" Layout
    End
End

Method "Input"
Begin
    Group
    Begin
        Input Data "Enter starting initial bet"
           to Record "amount" Data

        Input Data "Enter minimum table limit"
           to Record "minimum" Data
    End
End
```

Reverse D'Alembert System

The reverse D'Alembert system is sometimes called contra-Alembert. This system was developed for players who couldn't consistently win with the regular system. The betting progression system is in the reverse of D'Alembert where after each loss, one unit is subtracted from the next bet and after each win, one unit is added to the next bet. Let's take an example of how this type of betting system works.

BET #	UNITS BET	RESULT	NET GAIN
1	5	lose	-5
2	4	win	-1
3	5	lose	-6
4	4	win	-2
5	5	win	+3
6	6	lose	-3
7	5	win	+2

This method builds on winning streaks instead of chasing losses. The problem can occur is when to abandon a winning streak as you can see you would eventually hit the minimum table limit. As you can see from the example above, there are 4 wins and 3 losses. Again, this type of system is best when played on the even chance layouts. The reverse D'Alembert system that is created in this book will use the **follow the color** option of red and black and will stop after having four wins in a row or if you hit the table minimum. The algorithm is written like this:

When a new session is started, do this…

- Set the initial betting amount of 5 to a data record **amount**
- Set the initial table minimum limit of 1 to data record **minimum**
- Call **Input** routine to allow user to select an initial amount to bet and table minimum bet

On every spin, check for…

- On a win, increase bet by 1 by adding 1 to data record **amount**
    - Check if minimum table bet has been reached or there are 4 wins in a row, stop session.
- On a loss, decrease bet by 1 unit by subtracting 1 from data record **amount**
    - Check if **Bankroll** has been depleted and if so, stop session
- Copy the last color that has appeared to data record **color**
- Place a bet on data record **color** at using the data record **amount**
- If 0 appears, bet on last color

Method "Input" routine here…

- Input "Enter starting initial bet" (default is 5)
- Input "Enter minimum table bet" (default is 1)

The entire system written in RX Scripting is shown on the next page.

Reverse D'Alembert system written in RX Scripting from the algorithm above.

```
System "Reverse D'Alembert"
{
Uses the reverse D'Alembert progression system for the
"follow the color" session
}
Method "main"
Begin
   While Starting a New Session
   Begin
     Put 5 on Record "amount" Data
     Put 1  on Record "minimum" Data

     Call "Input"
   End

   While on Each Spin
   Begin
      If Any Even Bet has Won Each time
      Begin
         Add 1 on Record "amount" Data

         If Record "amount" Data < Record "minimum" Data
         Or Record "color" Layout has won 4 times in a row
         Begin
            Stop Session
         End
      End

      If Any Even Bet has Lost Each time
      Begin
         Subtract 1 to Record "amount" Data

         If Bankroll <= 0
         Begin
            Stop Session
         End
      End

      Copy Last Red-Black to Record "color" Layout

      Put 100% of Record "amount" Data
            to Record "color" Layout
    End
End

Method "Input"
Begin
    Group
    Begin
        Input Data "Enter starting initial bet"
           to Record "amount" Data

        Input Data "Enter minimum table limit"
           to Record "minimum" Data
    End
End
```

Labouchere System

The Labouchere system is also known as a cancellation system. Like the Martingale, it is a progressive method of betting. However, it is unlikely you will run up against the table limit. The Labouchere is a complicated method and requires the use of a pencil and paper while visiting land based casinos. Of course, when using Roulette Xtreme software, those items are not needed. Again, this system is mostly used on the even chance layouts. The system starts with an arbitrary line of numbers such as 1-1-2-3. The initial bet is the sum of the first and last numbers in the line. In this case: 4. If the initial bet wins, the first and last numbers in the line are cancelled (crossed off) leaving numbers 1-2. The next bet will be 3 units (adding the first and last numbers in the line). If the bet loses, the sum of 3 is then added to the end of the line as such: 1-2-3. Then the next bet repeats by taking the sum of the first and last numbers which is 4 and so on until the entire line is cancelled (crossed off). At which this session is ended and a new line is started. The total profit made when the entire line is cancelled is the sum of the original starting line. In this case, it would be 7 units. Let’s take an example of how this type of betting system works.

BET LINE	UNITS BET	RESULT	NET GAIN
1-1-2-3	4	lose	-4
1-1-2-3-4	5	lose	-9
1-1-2-3-4-5	6	lost	-15
1-1-2-3-4-5-6	7	win	-8
1-2-3-4-5	6	win	-2
2-3-4	6	lose	-8
2-3-4-6	8	lose	-16
2-3-4-6-8	10	win	-6
3-4-6	9	win	+3
4	4	lose	-1
4-4	8	win	+7

In the example above, the last line cancelled with a net gain of 7 units which is the sum of the numbers from the original line. The more important note here is that the net gain was obtained after six losses and five wins. Clearly a distinct advantage over the D’Alembert system. You should also note that when the line got down to 4, the bet was also 4. The rule is in this method: when the line is reduced to a single number, bet only with that number. The starting line can be of any length and contain numbers of any values. The larger the line, the more aggressive the session. Therefore, it is good practice to keep your starting line small as not to risk betting large bets at once. Having small lines with small wins is your best attack for this system. For example:

MILD	AGRESSIVE
1-1	1-2-3-4
1-2	5-1-10-12-5
1-1-1	4-4-5-6-1-10

The Labouchere system that is created in this book will perform by placing bets on the even chance layout (user choice) and the starting line will be 1-1. The session will end after the bankroll has been depleted and the session will reset after the line 1-1 is completely cancelled (crossed-off). Note that this system is using multiple call routines to help organize the system. The algorithm is written like this:

<u>When a new session is started, do this…</u>

- Call **Initialize** routine to set starting line to 1-1
- Call **Input** routine to allow user to select layout type
- Call **Place Bets** routine to make the initial bet

<u>On every spin, check for…</u>

- On a win, call **Remove Line** routine to remove first and last number in the line
    - Check if the entire line is cancelled and if so, display message and call **Initialize** routine to set starting line to 1-1.
- On a loss, call **Append Line** routine to add the last bet to the end of the line
    - Check if **Bankroll** has been depleted and if so, display a message and stop session
- Call **Place Bets** routine to make another bet

<u>Method "Place Bets" routine here…</u>

- Take the sum of the first and last number in the line of data record **line** and store it to a data record **sum**
- Place a bet from the data record **sum** to the even chance layout stored in the data record **even chance**

<u>Method "Remove Line" routine here…</u>

- Cancel the first and last number in the line of data record **line**

<u>Method "Append Line" routine here…</u>

- Append the last bet amount to the end of the line of data record **line**

<u>Method "Initialize" routine here…</u>

- Set line to 1-1 on data record **line**

<u>Method "Input" routine here…</u>

- Input "Select even chance layout type of: (Red, Black, Odd, Even, High, Low)
- Copy this layout type to a data record called **even chance**

You will see in the system on the next page that using multiple call routines make the system easier to understand.

Labouchere system written in RX Scripting

```
System "Labouchere"
{
  Labouchere system with starting line @ 1-1
}
Method "main"
Begin
   While Starting a New Session
   Begin
     Call "Initialize"
     Call "Input"
     Call "Place Bets"
   End

   While on Each Spin
   Begin
      If Any Even Bet has Won Each time
      Begin
         Call "Remove Line"

         If Record "line" Data Count = 0
         Begin
            Display "Line Complete"
            Call "Initialize"
         End
      End

      If Any Even Bet has Lost Each time
      Begin
         Call "Append Line"

         If Bankroll <= 0
         Begin
            Display "Bankroll Depleted"
            Stop Session
         End
      End

      Call "Place Bets"
    End
End

Method "Place Bets"
Begin
// get the first number and store to record "sum"
    Put 1 on Record "line" Data Index
    Put 100% of Record "line" Data to Record "sum" Data

// if more than 1 number in line, add last number to "sum"
    If Record "line" Data Count>1 Begin
        Set Max to Record "line" Data index
        Add 100% of Record "line" Data to Record "sum" Data
    End

// place a bet of the sum number to even chance layout
    Put 100% of Record "sum" Data to
         Record "even chance" Layout
End
```

System continued from previous page

```
    Method "Remove Line"
    Begin
    // remove the last number by using the move list down
        Move List Down by 1 on Record "line" Data

    // remove the first number by using the move list up twice
        Move List Up by 2 on Record "line" Data
    End

    Method "Append Line"
    Begin
    // point to the end of the line using max
        Set Max to Record "line" Data Index

    // set the pointer past the end of the record
        Add 1 to Record "line" Data Index

    // store the last bet to the end of the list
        Put 100% of Record "sum" Data to Record "line" Data
    End

    Method "Initialize"
    Begin
    // initialize the list to 1-1
         Set List [1,1] to Record "line" Data
    End

    //select which even chance layout to use
    Method "Input"
    Begin
        Input Dropdown "Select even chance layout type
                1:=Red
                2:=Black
                3:=Odd
                4:=Even
                5:=High
                6:=Low" to Record "type" Data

        If Record "type" Data = 1 then Begin
           Copy Red to Record "even chance" Layout End
        If Record "type" Data = 2 then Begin
           Copy Black to Record "even chance" Layout End
        If Record "type" Data = 3 then Begin
           Copy Odd to Record "even chance" Layout End
        If Record "type" Data = 4 then Begin
           Copy Even to Record "even chance" Layout End
        If Record "type" Data = 5 then Begin
           Copy High to Record "even chance" Layout End
        If Record "type" Data = 6 then Begin
           Copy Low to Record "even chance" Layout End
    End
End
```

Inside Number System

The Inside Number system in this book is designed to target a single number particular a number that is overdue. However, for this system to work effectively and stretch your bankroll as far as possible, the system starts by placing bets on the dozen layout of the target number. Then after a couple of spins with no hits, progress to the line layout, followed by the corner bet, the street, the split and finally the target number. The progression is as follows – assuming target number **26**.

BET #	BET TYPE	PAYS	BET AMOUNT	NET LOSS	NET WIN
1	Dozen	2	1	1	2
2	Dozen	2	1	2	1
3	Line	5	1	3	3
4	Line	5	1	4	2
5	Line	5	1	5	1
6	Corner	8	1	6	3
7	Corner	8	1	7	2
8	Corner	8	1	8	1
9	Street	11	1	9	3
10	Street	11	1	10	2
11	Street	11	1	11	1
12	Split	17	1	12	6
13	Split	17	1	13	5
14	Split	17	1	14	4
15	Split	17	1	15	3
16	Split	17	1	16	2
17	Split	17	1	17	1
18	#26	35	1	18	18
19	#26	35	1	19	17
20	#26	35	1	20	16
21	#26	35	1	21	15
-	-	-	-	-	-
-	-	-	-	-	-
32	#26	35	1	32	4
33	#26	35	1	33	3
34	#26	35	1	34	2
35	#26	35	1	35	1
36	#26	70	2	37	35

Notice at spin # 36, the system doubles the bet. The doubling continues if you do not win every 35 spins. As you can see, there is a chance of hitting the table maximum in about 315 spins. Clearly you have a good chance of winning before the 315th spin but if this doesn't happen then you will loose about 547 units. It is better to have a stop loss to prevent such a tragedy.

This system has several user inputs that allow you to enter several items such as: starting bankroll, win goal, stop loss and the number of sessions to play. Set these to match your comfort level when playing this system with any online casino game.

The algorithm is written like this:

When a new session is started, do this…

- Set roulette table layout to single wheel (0)
- Call **Initialize** routine to setup variables
- Call **Input** routine to allow user to make input selections

On every spin, check for…

- Increment data record **Spin Counter** and data record **Multiplier Counter** by 1
- Call **Check for loss** routine to see if the system is below the stop loss
- Call **Check for win** routine to see if the system achieved its win goal
- If the flag is false for **Ready to Bet** then do the following…
  - Copy the last number to a data record **Target Number**
  - If the last number is 0, then skip and wait for another spin else set the flag **Ready to Bet** to true
- If the flag is true for **Ready to Bet** then do the following…
  - Call **Place Bets** routine to make a bet on the layout determine by the progression count.

Method "Check for loss" routine here…

- If any outside or inside bet has lost, do the following…
  - If the bankroll is less that the stop loss
    - Display a message, end the session and exit program

Method "Check for win" routine here…

- If any outside or inside bet has won, do the following…
  - If there are remaining sessions to play, do the following…
    - Decrement data record **Sessions** by 1
    - If data record **Sessions** is <= 0 the
      - Display a message, end the session and exit the program
  - If the Bankroll is greater than win goal
    - Display a message, end the session and exit the program
  - Call **Initialize** routine to setup variables and start over

Method "Place Bets" routine here…

- If data record **Spin Counter** is between 1 and 2, do the following…
    - Place a bet reference by **bet amount** to the data record **Target Number** nearest Dozen layout
- If data record **Spin Counter** is between 3 and 5, do the following…
    - Place a bet reference by **bet amount** to the data record **Target Number** nearest Line layout
- If data record **Spin Counter** is between 6 and 8, do the following…
    - Place a bet reference by **bet amount** to the data record **Target Number** nearest Corner layout
- If data record **Spin Counter** is between 9 and 11, do the following…
    - Place a bet reference by **bet amount** to the data record **Target Number** nearest Street layout
- If data record **Spin Counter** is between 12 and 17, do the following…
    - Place a bet reference by **bet amount** to the data record **Target Number** nearest Split layout
- If data record **Spin Counter** is >= 18, do the following…
    - If data record **Multiplier Counter** is > 35
        - Reset the data record **Multiplier Counter** to 0
        - Double the data record **bet amount**
    - Place a bet reference by data record **bet amount** to the data record **Target Number** layout

Method "Initialize" routine here…

- Clear contents of data record **Target Number**
- Initialize data record **bet amount** to 1
- Initialize data record **Spin Counter** to 1
- Initialize flag **Ready to Bet** to false

Method "Input" routine here…

- Display message "For Single Zero wheel"
- Input your starting bankroll
- Input your win profit
- Input you stop loss
- Input how many sessions to play
- Add Bankroll to data record **Win** and data record **Stop** loss

The actual system **Inside Numbers** is shown on the next few pages.

Inside Numbers system (continues on next 3 pages)

```
System "Inside Numbers"
{
  Take the last number, bet on its layout in this order
  Dozen for 2 times, Line  for 3 times, Corner for 3 times,
  Street for 3 time ,Split for 6 times, Straight-up
  until a win (doubling bet every 35 spins)

  On win, reset and take the next last number, repeat above
}
Method "main"
Begin
    While Starting a New Session
    Begin
        Load Single Wheel
        Call "Initialize"
        Call "Input"
    End

    While on Each Spin
    Begin
        Add 1 to Record "Spin Counter" Data
        Add 1 to Record "Multiplier Counter" Data

        Call "Check for loss"
        Call "Check for win"

        If Flag "Ready to Bet" is False
        Begin
            Copy Last Number to Record "Target Number" Layout

            If Record "Target Number" Layout not = Number 0
            Begin
                Set Flag "Ready to Bet" to True
            End
        End

        If Flag "Ready to Bet" is True
        Begin
            Call "Place Bets"
        End
    End
End

Method "Check for loss"
Begin
    If Any Inside Bet has lost each time
    Or Any Outside Bet has lost each time
    Begin
        If Bankroll<= Record "Stop" Data
        Begin
            Display "You have reached your Loss target.
                     System will End"
            Stop Session
            Exit
        End
    End
End
```

System continued from previous page

```
Method "Check for win"
Begin
    If Any Inside Bet has won each time
    Or Any Outside Bet has won each time
    Begin
        If Record "Sessions" Data Not = 0
        Begin
            Subtract 1 from Record "Sessions" Data

            If Record "Sessions" Data <=0
            Begin
                Display "You have completed all sessions.
                         System will End"
                Stop Session
                Exit
            End
        End

        If Bankroll>= Record "Win" Data
        Begin
            Display "You have reached your Win goal.
                     System will End"
            Stop Session
            Exit
        End

        Call "Initialize"
    End
End

Method "Place Bets"
Begin
    If  Record "Spin Counter" Data >=1
    And Record "Spin Counter" Data <=2
    Begin
        Put 100% of Record "bet amount" Data to Dozen nearest
            of Record "Target Number" Layout
    End

    If  Record "Spin Counter" Data >=3
    And Record "Spin Counter" Data <=5
    Begin
        Put 100% of Record "bet amount" Data to Line nearest
            of Record "Target Number" Layout
    End

    If  Record "Spin Counter" Data >=6
    And Record "Spin Counter" Data <=8
    Begin
        Put 100% of Record "bet amount" Data to Corner
           nearest of Record "Target Number" Layout
    End
```

System continued from previous page

```
        If  Record "Spin Counter" Data >=9
        And Record "Spin Counter" Data <=11
        Begin
            Put 100% of Record "bet amount" Data
               to Street nearest of Record "Target Number" Layout
        End

        If  Record "Spin Counter" Data >=12
        And Record "Spin Counter" Data <=17
        Begin
            Put 100% of Record "bet amount" Data to Split nearest
               of Record "Target Number" Layout
        End

        If  Record "Spin Counter" Data >=18
        Begin
            If Record "Multiplier Counter" Data >35
            Begin
                Put 0 on Record "Multiplier Counter" Data
                Multiply 2 to Record "bet amount" Data
            End

            Put 100% of Record "bet amount" Data to Record
                "Target Number" Layout
        End
    End

    Method "Initialize"
    Begin
        Clear Record "Target Number" Layout
        Put 1 on Record "bet amount" Data
        Put 0 on Record "Spin Counter" Data
        Set Flag "Ready to Bet" to False
    End

    Method "Input"
    Begin
        Put 1 on Record "Sessions" Data
        Put 10 on Record "Win" Data
        Put 50 on Record "Stop" Data
        Put 100% of Bankroll to Record "Bankroll" Data

        Group
        Begin
            Display "For Single Zero wheel"
            Input Data "Enter your starting Bankroll"
                to Record "Bankroll" Data
            Input Data "Enter your Win Profit units" to
                Record "Win" Data
            Input Data "Enter your Stop Loss units" to
                Record "Stop" Data
            Input Data "Enter how many sessions to play
                        0=unlimited" to Record "Sessions" Data
        End
```

System continued from previous page

```
        Put 100% of Record "Bankroll" Data to Bankroll
        Add 100% of Bankroll to Record "Win" Data

        //Setup the Stop Loss by taking Bankroll-Stop
        //= Lowest Bankroll to keep
        Put 100% of Bankroll to Record "temp" Data
        Subtract 100% of Record "Stop" Data to Record "temp" Data
        Put 100% of Record "temp" Data to Record "Stop" Data
    End
```

RX SCRIPTING TEMPLATE

6

Do you create several systems and you noticed that you always create the same method routines such as placing a bet, checking for losses, asking for data inputs? You can create your system faster by using a template that provides most of the methods you need.

This chapter includes a template that you can use when creating a system and I will also show you how you can make this template be the default every time you create a new system when using the System Editor from Roulette Xtreme software.

Template to use when creating new systems

```
System "My System"
{
  Comment section
}
Method "main"
Begin
    While Starting a New Session
    Begin
        Call "Initialize"
        Call "Input"
    End

    While on Each Spin
    Begin
        Call "Check for Loss"
        Call "Check for win"
        Call "Place Bets"
    End
End

//routine to deal with any losses
Method "Check for Loss"
Begin
End

//routine to deal with any wins
Method "Check for win"
Begin
End

//routine to place bets on layouts
Method "Place Bets"
Begin
End

//routine to initialize the system
Method "Initialize"
Begin
End

//routine to ask for any data inputs
Method "Input"
Begin
End
```

As you can see, this template contains methods that you can use to check for losses, wins, place bets, initialize data and ask for inputs. However, this template is one example. You can always add or subtract from this template any method routines that you commonly use throughout your system designs.

On the next page, I'll show you how easily you can add this to Roulette Xtreme System Editor so every time you select **File -> New**, this template will be added.

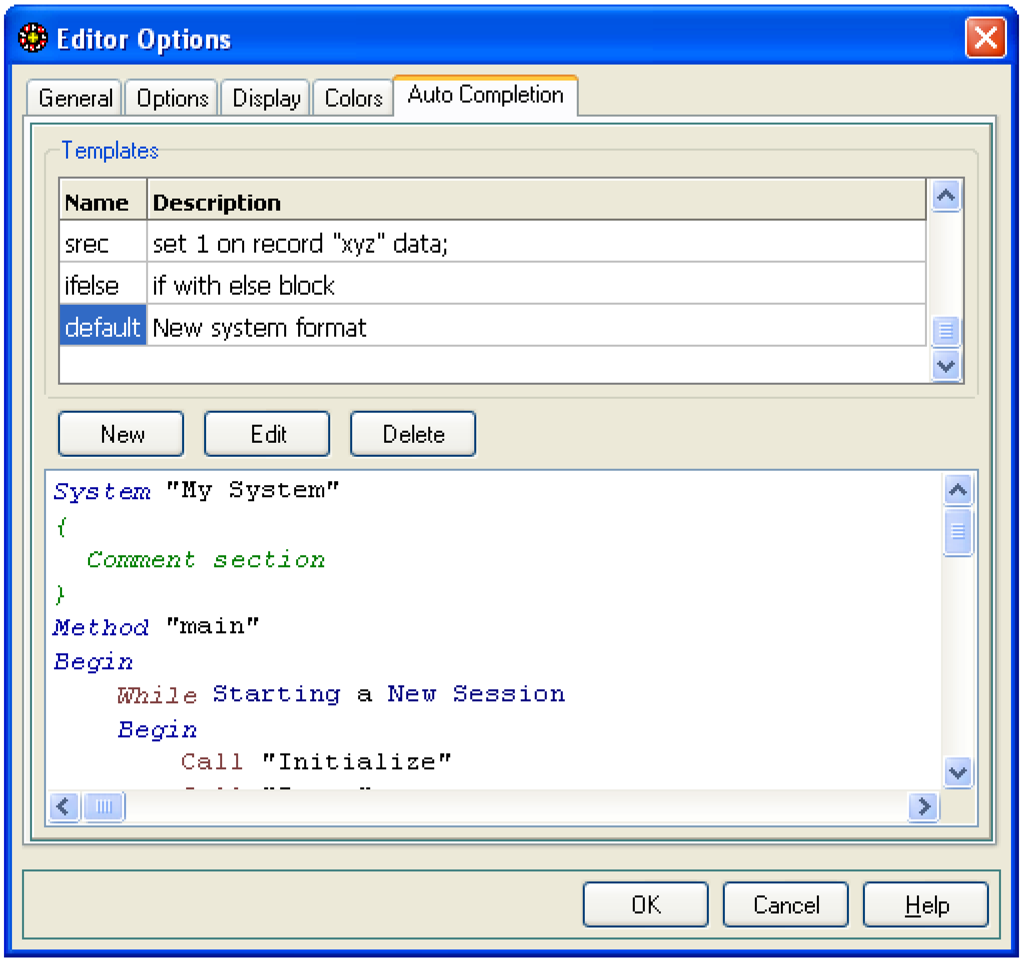
Adding the Template to System Editor

To add this template to the System Editor, perform the following steps:

1. Open the System Designer by selecting **Designer** -> **System Designer** from the main screen of Roulette Xtreme.
2. Within the System Designer, select **Options** -> **Editor Options** from the menu.
3. Click on the **Auto Completion** tab of the options screen.
4. Scroll down until you find the **default** template and click on that line.
5. Click inside the system editor screen just below New, Edit and Delete buttons.
6. Copy/Paste the template from the previous page into the system editor screen.
7. Click Ok to close the options screen.

Now every time you select **File** -> **New**, this template will be added to the system editor ready for you to start creating systems without the need to create the same methods over and over.

See screen below of the **Auto Completion** tab with this template added.



Name	Description
srec	set 1 on record "xyz" data;
ifelse	if with else block
default	New system format

```
System "My System"
{
  Comment section
}
Method "main"
Begin
    While Starting a New Session
    Begin
        Call "Initialize"
```