Rex++ Language System Programming Reference

Introduction

Welcome to the Rex++ Programming System complete with an easy to use integrated development environment (IDE). Rex++ is a programming language that is similar to the high level programming language BASIC (Beginners all-purpose Symbolic Instruction Code). You edit and run your programs using the built-in source code editor/interpreter all from within from the IDE.

Rex++ IDE

You start Rex++ by typing REX++ followed by the ***Enter*** key at the command prompt inside the Rex Blade Virtual Computer Interface (of course you will need a password!)

After the program loads you will see a screen that has four basic parts. The very top line is the ***editor status line***. This line displays the current line and column positions of the text as you type as well as the name of the file you are editing. Also displayed is the current ***edit mode***. In ***Insert*** mode, text is moved over to the right as you type and in ***Overstrike*** mode, text is written over at the current cursor position.

The second area of the screen is the ***text edit window***. In this area you can enter text as you would with any editor. About 16 pages of source code and can be entered with a line width of 50 characters per line. To end a line simply press the ***Enter*** key. Below is a summary of the commands supported by the Rex++ Code Editor:

Cursor movement commands:

Character left	[LEFT]
Character right	[RIGHT]
Word left	[Ctrl+LEFT]
Word right	[Ctrl+RIGHT]
Line up	[UP]
Line down	[DOWN]
Beginning of line	Home
End of line	End

Insert and delete commands:

Delete character	Del
Delete character to left	Backspace
Delete line	Ctrl+Y
Insert mode on/off	Ins

The third area of the screen is the ***message status line***. Any informative messages displayed by the program will be shown on this line.

The last area of the screen is the ***command line.*** Below is a summary of the available commands:

command line options:

F2 Loads a source file from disk. A list of all the Rex++ programs in the current sub directory is displayed in a pick list. Use the arrow keys to highlight the file and ***Enter*** to select it.
F3 Saves the current source file to disk.
F4 Runs the current source file in the editor window.
F5 Clears the current source file in the editor window.
F6 Exit Rex++ and returns back to the Virtual Computer Interface of Rex Blade.

Language Guide

This section presents the formal definition of the Rex++ language.

Statements

Rex++ is made of up of a series of statements that describe the actions the program can take. These are examples of statements:

```
a := b + c;
Print("this is a test");
if (x < 2)
  Answer := x * y;
endif;
```

Simple statements can either assign a value or transfer the running of the program to another statement in the code. The first two examples shown in the examples are simple statements.

Structured statements can be compound statements that contain multiple statements, conditional and repetitive statements that control the flow of logic within a program.

Expressions

Just as a sentence is made up of phrases, so is a Rex++ statement made up of expressions. The phrases of a sentence are made up of words, and the expressions of a statement are composed of elements called **factors** and **operators**. Expressions usually compare things or perform **arithmetic**, **logical** or **boolean** operations. Lets look at some examples of expressions:

x + y	**A simple sum.**
Done <> Error	**A NOT-EQUAL comparison.**
i <= Width	**A LESS THAN OR EQUAL comparison.**
-n	**The opposite of the variable n.**

Tokens

Tokens are the smallest meaningful elements in a Rex++ program. They makeup the factors and operators of expressions. Tokens are special symbols, **reserved words**, **identifiers**, **labels**, **numbers** and **string contestants**; they are akin to the words and punctuation of a written human language. These are examples of Rex++ tokens:

(	**Left Paren, usually used for grouping.**
:=	**Assignment operator.**
print	**A language keyword.**
;	**The end of line character.**

Operators

Operators are classified as arithmetic operators, logical operators, string operators and relational operators.

Table 1.0 below illustrates the types of operands and results for binary and unary arithmetic operations.

Table 1.0 - Operators

Arithmetic operators:

Operator	Operation	Operand types	Result type
+	addition	number type	number type
-	subtraction	number type	number type
*	multiplication	number type	number type
/	division	number type	number type

Unary arithmetic operations:

Operator	Operation	Operand types	Result type
+	sign identity	number type	number type
-	sign negation	number type	number type

Boolean operators:

Operator	Operation	Operand types	Result type
and	logical and	Boolean	Boolean
or	logical or	Boolean	Boolean

String operator:

Operator	Operation	Operand types	Result type
+	concatenation	string type	string type

Rex++ allows the + operator to be used to concatenate two strings operands. The result of the operation S + T, where S and T are of type string. If the resulting string is longer than 255 characters, it's truncated after character 255.

Relational operators:

Operator	Operation	Operand types	Result type
=	equal	number, string types	Boolean

<>	not equal	number, string types	Boolean
<	less than	number, string types	Boolean
>	greater than	number, string types	Boolean
<=	less than or equal to	number, string types	Boolean
>=	greater or equal to	number, string types	Boolean

Variables

A variable can hold a value that can change. Every variable must be a **type**. A variable's type specifies the set of values the variable can have. Rex++ support two types of variables, **numbers** and **strings**. A number can be any **integer** or **real** (decimal) number and a string variable can be up to 255 characters.

For example, this next program declares that variables x and y are of type number; therefore, the only values x and y can contain are numbers. Rex++ displays error messages if your program tries to assign any other type of value to these variables.

```
var x: number;    {variables x is type number}
var y: number;    {variable y is type number}

x := 12;
y := 10;
x := x + y;
```

x is assigned the value 12 originally; two statements later it is assigned the a new value, x + y. As you can see, the value of a variable can vary.

All variables are declared with the **var** statement. Variables can be declared anywhere in the program, but they must be declared first before use.

Identifiers

Identifiers denote **constants**, **types**, **variables** or **commands**. An identifier can be up to 63 characters and must begin with a letter or an underscore character "_" and can not contain spaces. **Letters**, **digits**, and **underscore** characters are allowed after the first character. Identifiers are **not** case sensitive.

Numbers

Ordinary decimal notation is used for numbers that are constants of integer and floating point types. Numbers with decimals or exponents denote floating point-type constants, while other decimal numbers denote integer-type constants; they must be within the range -2,147483 to 2,147483.

Character strings

A character string is a sequence of zero or more characters from the extended ASCII character set, written on one line in the program and enclosed by double qoutes. A character string with nothing between the double quotes is a **null** string. Examples of character strings include:

“Xtreme Games”	**{ Xtreme Games }**
“This is a test’	**{ This is a test }**
“”	**{ null string }**
“ “	**{ a single space }**

Comments

The following constructs are comments and are ignored by Rex++:

```
{ Any text not containing right brace }
```

Library reference

This section contains detailed descriptions of all the Rex++ procedures and functions with examples.

Declaration
Cls (color: number);

Purpose
Clears the screen.

Description
Cls will clear the screen with the specified color value. The color can range for 0 to 255.

Example
```
Cls(5);
```

Declaration
Delay (time: number);

Purpose
Delays program execution.

Description
Delay stops the currently running program for a specified time period is milliseconds. The time can range from 0 to 32768.

Example
```
var index: number;
var color: number;

randomize;

loop(index=1 to 10)
  number := rand(1, 256);
  println("number =", number);
  Delay(10); {delay program for ten milliseconds}
endl;
```

Declaration
End;

Purpose
Ends the currently executing program.

Example
```
var n: number;

n := 10;
if (n = 10)
  println("n is 10");
  end; {stop program}
endl;
println("Nope, n is not 10");
```

Description

End will stop program execution on the line containing *End*. The command can be used to abort the program based on some condition.

Declaration
Endif;

Purpose
End the code black started with a If command

Description
When a *If* command begins execution, all commands between *If* and *Endif* will be executed over and over until the If expression becomes true. Endif tells Rex++ where this block of code ends. There must be a Endif for every If and an error will be displayed if they are unbalanced.

Example
```
var n: number;

n := 20;

if (n=20)
  println("n is equal to 20");
endif; {ends the if block}
```

Declaration
Endl;

Purpose
End the code block started with a *Loop* command.

Description
When a *loop* command begins execution, all commands between *Loop* and *Endl* will be executed over and over until the loop expression becomes true. *Endl* tells Rex++ where this block of code ends. There must be a *Endl* for every Loop and an error will be displayed if they are unbalanced.

Example
```
var n: number;

loop(n = 1 to 10)
  println(n);
endl; {ends the loop statement}
```

Declaration
Goto $lable

Purpose
Goto will transfer execution of the program to a specific location.

Description
Goto is a control statement that transfers execution of the program to the instruction following a **label.** All labels in Rex++ must begin with a $ (dollar) symbol.

Example
```
var x: number;

x := 1;

if (x = 1)
  println("x is one.");
  if (x < 2)
    goto $less;
endif;
```

```
endif;

$less
println("x was less than 2");
```

Declaration
If (expression: Boolean)

Purpose
Selects execution of program based on a condition.

Description
The *If* command is used to conditionally execute component statements in the program. The *If* expression must evaluate to ***True***, the commands following the *If* will execute until *Endif* is encountered. If the expression produces ***False***, execution starts on the line following the paired *Endif*. Note: *If/Endif* must be balanced or a run-error will be displayed.

Example
```
  var n: number;

  randomize;
n := rand(1, 10);
if (n < 5)
  println("n is less than 5);
endif;
```

Declaration
Kbhit: boolean;

Purpose
Checks if a key is pressed.

Description
Kbhit checks to see of a key has been pressed. *Kbhit* returns a boolean expression of TRUE if a key has been hit and FALSE if not.

Example
```
  if (kbhit)
    println("a key was hit…");
  endif;
```

Declaration
Kbcode(code: number): boolean;

Purpose
Checks if a specific key is pressed.

Description
Kbcode allows you to check if a specific key is pressed. It has the advantage of allowing you to detect multiple keystrokes. Pass to *Kbcode*, the scan code of the key you are detecting and it will return TRUE if the key is being pressed or FALSE if not.

Example

```
  if (kbcode(1))
    println(“the ESC key was pressed.”);
  endif;

  if (kbcode(72))
    println(“the UP arrow was pressed.”);
```

Declaration
Line(x1:number, y1: number, x2:number, y2: number);

Purpose
The *Line* command plots a line using the current graphics color that was set with *SetColor.*

Description
Line can display a line on the screen from coordinates x1,y1 to x2,y2. The x coordinates can range from 0-319 and the y coordinates can range from 0-199. An error is returned if the coordinates are outside these values.

Example
```
var x1: number;
var y1: number;
var x2: number;
var y2: number;
var n: number;

randomize;

for (n= 1 to 1000)
  x1 := rand(0, 319);
  y1 := rand(0, 199);
  x2 := rand(0, 319);
  y2 := rand(0,199);
  setcolor(rand(1,  255));
  line(x1, y1, x2, y2);
endl;
```

Declaration
Loop(control variable:number = initial value TO final value: number)

Purpose
Loop specifies certain commands to be executed repeatedly.

Description
The *Loop* statement causes commands to be repeatedly executed while a progression of values is assigned to a control variable. The control variable must be a ***number*** type and already defined. The value of the control variable is incremented by one for each repletion. If initial value is greater or less than final value, the contained commands isn’t executed. Commands will executed down to the *Endl* command. There must be a matching *Endl* for every loop or and error will be returned.

Example
```
  var n: number;

  loop(n = 1 to 10)
    println(n);
  endl;
```

Declaration
Plot(x:number, y: number)

Purpose
Plots a single pixel on the screen using the current color.

Description

Plot will draw a pixel on the screen at the specified coordinate using the color set by the last *SetColor* command. The x value can range from 0-319 and the y value can range from 0-199. Any values outside these ranges results in a run-time error.

Example
```
plot(50, 50);
plot(160, 100);
```

Declaration

Print(expression:number|string|string constant)

Purpose

Outputs data to the screen with no line feed.

Description

Print can display a number, string and string constant (text between double quotes). *Print* does not move the cursor to the next line. *Print* will use the color set by the last call to *SetColor* and will be printed at the current cursor position.

Example
```
var n: number;
var s: string;

n := 100;
s := "Jarrod Davis"
print("This is a test");
print(n);
print(s);
print(n => 100); {will display TRUE}
print(n = 1); {will display FALSE}
```

Declaration

PrintLn;

Purpose

Outputs data to screen with line feed.

Declaration

Same as *Print* but performs a line feed and move the cursor to the beginning of the next line.

Declaration

Rand(min: number, max: number)

Example

See Print example.

Purpose

Returns a random number between min and max.

Description

Rand can be used to get a random number between minimum range (0) and maximum range (32767). Any values outside these ranges results in a run-time error..

Example
```
var n: number;
randomize;
```

```
n := rand(0, 10);
println(n);
```

Declaration
Randomize;

Purpose
Seeds the random generator.

Description
Randomize initializes the random generator. This statement must be called be using *Rand*.

Declaration
SetColor(color: number);

Purpose
Set a new color value.

Description
SetColor specifies a new color value that is used by all of the graphic routines that draws to the screen. The color value can range between 0 and 255. Any values outside this range will result in a run-time error.

Example
```
SetColor(50);
```

Declaration
SetCursor(x: number, y: number);

Purpose
Sets the current cursor position.

Description
SetCursor sets the new graphics cursor position. The x coordinate can range from 0 to 319 and the y coordinate can range from 0 to 199. Any values outside the ranges will result in a run-time error.

Example
```
SetCursor(50, 50);
```

Declaration
To

Purpose
Used only in the *Loop* command when specifying the initial and final values.

Description
To must be used to separate the initial and final values in the *Loop* command. A run-time error is returned is it is not found.

Example
```
var x: number;

loop(x = 1 TO 100);
  …
endl;
```

Declaration
Var

Purpose
Declares a variable.

Description
Var is used to declare a variable. Rex++ uses two types of variables, a number and a string. Variables have to be declared before used or a run-timer error will result.

Example
```
var n: number;
var s: string;

n := 1;
s := “test”;

println(n);
println(s);
```

Declaration
WhereX;

Purpose
Returns the current X coordinate of the graphics cursor.

Description
WhereX is a function that returns the current horizontal position of the graphics cursor. It will return a value in the range of 0 to 319.

Example
```
println(wherex);
```

Declaration
WhereY

Purpose
Returns the current Y coordinate of the graphics cursor.

Description
WhereY is a function that returns the current vertical position of the graphics cursor. It will return a value in the range of 0 to 199.

Example
```
println(wherey);
```

Examples

This section contains listings for a number of example programs that demonstrates the various features of the Rex++ language. All of these programs will be installed on your hard drive along with Rex Blade, so you don't have type them in. To load any of them, simply note their name(s) and use the LOAD command from the IDE.

The first example shown below in Listing 1.0 illustrates the use of the random number generator coupled with the clear screen function. The program begins by declaring a loop variable followed by seeding the random number generator. The program then enters the main loop which iteratively clears the screen with a new random color each cycle.

Listing 1.0 - CLS.RPP

```
{ This program demonstrates the cls command }

var x: number;               { declare x as a number }

randomize;                   { initialize random number generator }

loop(x=1 to 50)              { setup to loop from 1 to 50 }
  cls(rand(0, 255));         { clear the screen with a random color }
  delay(50);                 { pause program execution for 50 milliseconds }
endl                         { end the loop }
```

The next example program shown below in Listing 2.0 illustrates how to use the cursor setting and querying functions. The program sets the position of the cursor and then retrieves it with the "where" functions. This is a good example of how to track your text output for formatting.

Listing 2.0 - CURSOR.RPP

```
{ This program demonstrates the setcursor command }

var x: number;               { declare x as a number }
var y: number;               { declare y as a number }

setcursor(50, 80);         { set graphics cursor to position 50,80 }

println("");                 { print a blank line }
print("X = ",wherex);        { print value of current horizontal position }
print("Y = ",wherey);        { print value of current vertical position }
```

All computer languages allow complex expressions to be evaluated, however, Rex++ goes a little farther in some areas such as string processing. Listing 3.0 below shows both numeric and string data types benig defined and output. Notice the use of the addition operator to concatenate strings together, this is a very powerful language construct.

Listing 3.0 - EXPR.RPP

```
{ This program demonstrates expressions and strings }

var s: string;               { declare x as a string }
var n: number;               { declare n as a number }

n := 100 + 100;              { assign n a number value }
s := "This is a test" + " this is another test "; { assign s a string value }
println(s);                  { print value of s }
println(n);                  { print value of n }
println("I am the " + "Man!"); { print a string expression }
println( ((4*5) / 2) + (100/2)); { print a number expression }
```

The next example below, Listing 4.0 is our first program that does something a bit graphical. The program begins by declaring a number of working variables, the program then enters into a main loop and repeatedly computes a set of random numbers. These numbers are used as parameters to the color and line drawing functions. The result, a collection of random lines drawn on the screen in random colors.

Listing 4.0 - LINE.RPP

```
{ This program demonstrates the line command}

randomize;                     { setup the random number generator }

line(0,0,319,199);           { draw a diagonal line from 0,0 to 319,199 }


var n: number;                 { declare n as a number }
var x1: number;                { declare x1 as a number }
var y1: number;                { declare y1 as a number }
var x2: number;                { declare x2 as a number }
var y2: number;                { declare y2 as a number }

loop(n=1 to 1000)              { set n to loop from 1 to 1000 }
  x1 := rand(0, 319);          { assign x1 a random number from 0 to 319 }
  y1 := rand(0, 199);          { assign y1 a random number from 0 to 199 }
  x2 := rand(0, 319);          { assign x2 a random number from 0 to 319 }
  y2 := rand(0, 199);          { assign y2 a random number from 0 to 199 }
  setcolor(rand(1, 255);       { set graphics color a random color from 1 to 255 }
  line(x1, y1, x2, y2);        { draw line }
endl;                          { end loop }
```

Listing 5.0 below is similar to Listing 4.0 except that instead drawing random line segments, random pixels are drawn. However, don't be fooled by the example's simplicity since there is a more important language construct to be learned here. Notice that instead of using local variables to hold random data values, the random pixel postions are computed as parameters to the plot function. Hence, you may use this technique freely to save a local variable(s).

Listing 5.0 - PLOT.RPP

```
{ This program demonstrates the plot command }

var x: number;                 {declare x as a number}

randomize;                     {seed random number generator}

loop(x=1 to 10000)           { set x to loop from 1 to 1000 }
  setcolor(rand(1, 255));      { set graphics color a random number from 1 to 255 }
  plot(rand(0, maxx), rand(0, maxy)); { plot pixel at random x,y position }
endl;                          { end loop }

println("I am the man!");      { print a string expression }
```

No language would be complete without the ability to create text output. Rex++ has two functions to accomplish this task: *print()* and *println()*. Listing 6.0 below shows an example use of both of these functions, notice that the functions can take either numeric or string data.

Listing 6.0 - PRINT.RPP

```
{ This program demonstrates the print/println commands}

var x:number;                  { declare x as a number }
var y:string;                  { declare y as a string }

loop(x=1 to 10)                { setup x to loop from 1 to 10 }
 println("number: ", x);       { print value of x }
endl;                          { end loop }

print("this is a test");       { print string constant with no line feed}
println(" this is on the same line.");  { print string constant with line feed }
println("but this is not.");   { print string constant with line feed }
```

Listing 7.0 is a formal example of using the random functions, not too exciting!

Listing 7.0 - RAND.RPP

```
{ This program demonstrates the randomize/rand commands}

randomize;                     { setup random number generator }

var n: number;                 { declare n as a number }

loop(n=1 to 20)                { set n to loop from 1 to 20 }
  println(rand(1, 1000));      { print a random number between 1 and a 1000 }
endl;                          { end loop }
```

Listing 8.0 below illustrates the use of the real-time keyboard I/O functions which allow you to not only detect if a key has been pressed, but if multiple keys have been pressed. This particular example shows how to track the arrow keys.

Listing 8.0 - KEYTEST.RPP

```
{ This program demonstrates the kbcode function and the goto command }

println("Press the arrow keys, <ESC> to quit...");          { tell what to do }

setcolor(14);                                                { set text color to yellow }

$checkkeys                                                   { define a label }

  {clear keys}
  setcursor(0,6);
  print("           ");
  setcursor(0,12);
  print("           ");
  setcursor(0,18);
  print("           ");
  setcursor(0,24);
  print("           ");

  {check up arrow}
  if (kbcode(72))
    setcursor(0, 6);
    print("up arrow");
  endif;

  {check down arrow}
  if (kbcode(80))
```

```
   setcursor(0, 12);
   print("down arrow");
  endif;

  {check left arrow}
  if (kbcode(75))
    setcursor(0,18);
    print("left arrow");
  endif;

  {check right arrow}
  if (kbcode(77))
    setcursor(0,24);
    print("right arrow");
  endif;


  {check for <ESC>}
  if (kbcode(1))
    goto $out
  endif;

goto $checkkeys

$out
println("");
println("out of here");
```