The Sprite32! User's Manual

First Rate Embedded Microsystems

Revision 1.1

https://store.steampowered.com/app/2065990/FREM_Sprite32/

Table of Contents

Chapter 1: Introduction to the Sprite32!

Congratulations on your purchase of the Sprite32! Microcomputer, the premier home computer for the modern computing enthusiast.

Refer to this manual often to maximize your computing experience while using the Sprite32!

Cheers, and happy computing!
-- The FREM Team

Accessing the Manual

Press **F1** on the keyboard to show or hide this manual any time.

When using the computer in Sandbox, Tutorial, or Challenge mode, you can also press the *Manual* button located at the bottom right of the interface, just underneath the power controls.

Navigating the Manual

Use the chapter selection at the bottom of the window to choose from the various Manual sections. The **Next** and **Previous** buttons will navigate to the next and previous sections accordingly.

Press the Close (**F1**) button on the keyboard to close the Manual.

Shortcuts & Hot Keys

Throughout the manual, shortcuts and hotkeys for the host operating system will be used. Press **Menu (F2)** to view the full list of shortcuts for your operating system.

Running Code Examples

Text can be copied directly from this manual by highlighting it with the mouse and pressing **CTRL+C**. Paste it into the editor using **CTRL+V**. Keep this in mind when viewing syntax and trying out code samples!

You can also click links displayed near code examples to automatically insert snippets into the Code Editor.

Example:

```
WriteLn(
      "Click the [Add It] link above",
      "To add this code to the editor"
)
```

Chapter 2: Writing and Running Programs

FREMScript is a high-level programming language designed for the *FREM Sprite32!* microcomputer. Refer to the chapters of this manual for details about everything *FREMScript* has to offer.

Let's get started with some examples!

Your First Program

Start with a blank editor by pressing **CTRL+N** on the keyboard.

Type the following code into the editor:

```
Cls()
WriteLn("Hello Computer!")
```

Press **Close (F1)** to hide the manual, and then press **RUN (F5)** to execute the program. When you are finished, press F1 to return to the manual.

Try a few more lines of code out:

```
WriteLn(10 + 10)
WriteLn(100 - 200)
WriteLn(5 * 2)
WriteLn(100 / 50)
```

Using Comments

Use comments to annotate your code. Comments are discarded during evaluation and do not affect the program outcome. Comments are indicated with # character. When the parser encounters a #, it will ignore the remainder of a line.

Example:

```
# This is a full line comment

# Comments may span many lines
# by using multiple # characters.

# Comments can trail a line like so:
Write("Hello") # Print to Screen
```

Spacing and Capitalization

The *FREMScript* evaluator doesn't care how code is formatted. It is a common convention among programmers to use whitespace (spaces and tabs) to align code so that it is readable and easy to maintain.

FREMScript is case-insensitive. Instructions may be written in various ways:

```
# These all work.

WriteLn("Hello")
writeln("World")
WRITELN("How are you today?")
```

The following conventions are used throughout this manual for all *FREMScript* examples:

Keywords are always capitalized:

SET
READY
FUNCTION

Functions and built-ins are UpperCamelCase:

WriteLn()
IsSet()
SysPut()

Variable types are lowercase:

list
number
boolean
string
map
function

Operators are capitalized:

EQ
NEQ
GTE
LT
MOD

Going to Big Screen

Press **SCREEN SIZE (CTRL+=)** button to alternate between the large/small screen views.

Clearing Text

Use the **CLEAR (F7)** button on the keyboard to clear printed text from the screen at any time.

Try filling the screen with some text lines with the following program, then press CLEAR (F7) to clear them:

Example:

```
# Print several lines of text.
EACH line IN Range(0, 19) REPEAT
      WriteLn("Lines of text!")
DONE
```

Stopping a Program

For longer running programs, you may wish to end execution at times. Press the **HALT PROGRAM (F6)** button on the keyboard to stop any running program immediately.

Consider the following example code, which creates a loop that runs forever.

Example:

```
WHILE true REPEAT
      WriteLn("Loop!")
DONE
```

Pressing HALT (F6) while the above program is running will show a status of "TERMINATED" in the information window beneath the editor.

Restarting the Computer

The system can be rebooted at any time using the **RESET (F9)** button. Any active programs are immediately stopped and the system restarts and runs the reset program.

Resetting the system will leave many settings intact, such as colors, memory settings, and cache.

Power the Computer Off/On

The **POWER (F10)** button will cycle the power off/on. Power cycling the computer is the best way to get the Sprite32! back to its initial state, as all colors, memory settings, and cache will be erased.

The next chapter delves deeper into the syntax of *FREMScript*.

Chapter 3: Writing to the Text Layer

The **Text Layer** handles the display of all text on the Sprite32! It is the top-most display layer, meaning that it will always show above other layers when it is enabled.

Displaying lines

WriteLn() prints text on a new line. It accepts multiple values, and will print each value you give it on its own line.

Example:

```
WriteLn("Greetings.")
WriteLn("Hello", "User")
WriteLn(1, 2, 3)
```

Displaying values

Write() prints text on the active screen line. It is useful for concatenating text on the screen. It accepts multiple values, and will print each value you give it, appending each to the previous.

Example:

```
Write("Greetings.")
Write("Hello", " ", "User")
Write(1, 2, 3)
```

Clearing the Screen

To clear the screen of all text, use the Cls() command. This is equivalent to pressing the CLEAR (F7) button.

Example:

```
WriteLn("This text will be erased")
Cls()
```

Cls() is commonly used at the start of programs to clean up any content from previous program runs.

Maximum Line Length

The Sprite32! screen supports a maximum of 40 characters per line. Any text that exceeds the maximum line length will not be displayed.

Toggling the Display of the Text Layer

You can enable/disable the Text Layer to show or hide printed text.

To disable the Text Layer, leverage the System Boolean Register that manages it.

Example:

```
# Hide the Text Layer
SysPut("B", 10, false)

# Show the Text Layer
SysPut("B", 10, true)
```

You can learn more about the System Registers in Chapter 16.

When the text layer is disabled, you can still write to it. If you re-enable the text layer, any text that was printed to it will appear. In the following example, we disable the Text Layer, write some text to it, wait a few seconds, then enable the text layer.

Example:

```
# Hide the Text Layer
SysPut("B", 10, false)

WriteLn("This text will appear in a moment,")
WriteLn("Once the text layer is enabled.")

Wait(2)

# Show the Text Layer
SysPut("B", 10, true)
```

Chapter 4: Variables, Assignments, and Types

Variables

Variables define values for use in your programs. They are declared using the SET keyword, prefixed by a type that indicates what kind of value it holds.

Example:

```
SET number hoursInDay = 24
SET string FirstName = "John"
SET boolean powerIsOn = true
SET list oneTwoThree = [1, 2, 3]
SET map user = {
      "Name": "John",
      "Title": "Programmer"
}
```

Decimals can also be stored in the number type.

Example:

```
SET number pi = 3.14159
```

Note: The system supports up to 6 places of decimal precision for any value stored in a number.

Assignments

Assignment occurs when the variable is first defined using SET, as in the examples above. Anything to the right of the Equals (=) sign is the assignment. Variables can be reassigned by supplying a new value to the right of Equals (=) after the initial SET.

Example:

```
# First assignment of hours
SET number hoursInDay = 24
WriteLn(hoursInDay)

# Hours in a Martian day
hoursInDay = 25
WriteLn(hoursInDay)

# On Venus
```

```
hoursInDay = 5832
WriteLn(hoursInDay)
```

Types

FREMScript is a typed language, meaning that the evaluator checks the type assigned to a variable to decide how to use it. When assigning a variable of a particular type, it is locked to that type until it is **UNSET**. Attempting to mix types will result in error.

Example:

```
# a is a number
SET number a = 1234

# It can interact with other numbers
a = a + 10.5
SET number b = a + 7

WriteLn(a, b)

# C is a boolean
SET boolean c = true
b = a + c # incompatible types error

# Name is a string
SET string name = "John"
# We can append to name with "+"
name = name + " the Programmer"

# A list contains an array of items.
SET list nums = [1, 2, 3, 4, 5]

# A list can mix-and-match different
# types of values.
SET list things = [1, "two", 3.0]

# Maps "map" values to specific
# "keys".
SET map user = {
      "Name": "John",
      "Title": "Programmer"
}
```

```
# Functions are a built-in or user-defined
# function. Learn more about functions in
# Chapter 9.
SET function Print = WriteLn

# Now Print() is a clone of the
# WriteLn() function.
Print("Hello!")
```

You can use the built-in function *Dump()* to write the type and value of a variable to the screen.

Example:

```
SET number a = 123.456
Dump(a)

SET boolean hasPower = false
SET string name = "John"
Dump(hasPower, name)
```

Uninitialized Variables

Variables can be set without initial values by using the **READY** keyword.

Example:

```
READY number counter
READY string name
READY boolean powerIsOn
READY list myTodoList
READY map user
READY function Print
```

Variables declared with **READY** hold default values according to their type.

The rules are as follows:

number: 0
string: ""
boolean: false
list: []
map: {}

Deciding when to use READY depends on a particular application. Here are some examples.

Example:

```
# Counter is "0" by default
READY number counter
WHILE counter LT 10 REPEAT
      counter = counter + 1
DONE

# Result for math operation defaults
# to 0.
READY number total
total = 5 * 25
```

Deleting Variables

The **UNSET** keyword provides the mechanism for removing variables from the program. Once Removed, the variable can no longer be referenced for the remainder of the program, unless it is **SET** again.

Example:

```
SET number a = 10
WriteLn(a)

UNSET a
WriteLn(a) # Error, a not set
```

Check for Variables

The built-in function *IsSet()* provides the mechanism for checking whether a particular variable has been defined.

Example:

```
# A is defined
READY number a
WriteLn(IsSet("a")) # True

# B is not defined
WriteLn(IsSet("b")) # False

# A is no longer defined
UNSET a
```

```
WriteLn(IsSet("a")) # False
```

Variable Casing

Variables are case-insensitive in *FREMScript*. Uppercase, lowercase, mixed case variables all point to the same value.

Example:

```
SET number abc = 123

# These all refer to abc.
WriteLn(abc)
WriteLn(Abc)
WriteLn(ABC)

# Same for assignments.
aBC = 456
WriteLn(abC)
```

Assigning Values to Lists

List variables contain zero or more elements, which are values of any variable type (string, number, boolean, etc.). Lists are indexed numerically, starting with 0 for the first element.

Example:

```
SET list prices = [19.99, 29.99, 3.99]

# Show the first price in the list
WriteLn(prices[0])

# Show the last price in the list
WriteLn(prices[2])

Elements in lists can have mixed types:

SET list things = [123, "four", false]
```

Lists can even contain other lists:

```
SET list pairs = [
      [1, 2],
      [3, 4],
```

```
      [5, 6]
]
```

Like other variables, lists can be overwritten:

```
SET list nums = [1, 2, 3]
nums = [4, 5, 6]
```

Specific elements in a list can be overwritten by assigning values of the same type to their index.

```
SET list nums = [1, 2, 3]

# Overwrite the second element
nums[1] = 999

Dump(nums)
```

Modifying Lists with Functions

There are several built-in functions for adding elements to lists.

Append

Append adds one or more values to the end of a list. Each specified value is added to the end of the list, one at a time.

Example:

```
READY list inventory
Append(inventory, "sword", "shield")
```

Prepend

Prepend adds one or more items to the beginning of a list. Each specified value is added to the front of the list, one at a time.

Example:

```
Prepend(inventory, "potion", "helmet")
```

Size

Get the total number of elements in a list with the Size() function.

Example:

```
SET number count = Size(inventory)
```

There are also built-in functions for removing elements from lists.

Shift

Shift removes the first item from a list and outputs it. You can choose to assign it to a variable.

Example:

```
SET string first = Shift(inventory)
```

Pop

Pop removes the last item from a list and outputs it.

Example:

```
SET string last = Pop(inventory)
```

Remove

Remove unsets values at specific index positions. Unlike Shift and Pop, Remove does not output anything.

Example:

```
SET list a = [1, 2, 3]

# Remove the second item in the list
Remove(a, 1)

# Value "2" has been removed.
Dump(a)
```

Assigning Values to Maps

Maps store data using index/value pairs. This is similar to how lists work, except that instead of number indexing, a map uses string indexing.

Here's how to declare a map:

```
SET map employee = {
      "name": "Sam",
      "title": "Programmer"
```

```
}
```

Access the values by index:

```
WriteLn(employee["name"] ,employee["title"])
```

Overwrite map values:

```
employee["title"] = "Sr. Programmer"
```

Index

To add new index/value pairs to maps, use the Index() function.

Example:

```
Index(employee, "age", 28)
WriteLn(employee["age"])
```

Remove

Unset index/value pairs in a map using the Remove() function:

Remove(employee, "age")

Iterating over map values

Direct iteration of maps is not supported. Use the GetKeys() built-in function to get a list of keys as strings, then iterate through the map using the list.

Example:

```
SET list keys = GetKeys(employee)
EACH key in keys REPEAT
      WriteLn(employee[key])
DONE
WriteLn(employee["age"])
```

Chapter 5: Arithmetic Operations

FREMScript supports arithmetic operations through the use of "operator" symbols and phrases:

Addition: +
Subtraction: -
Multiplication: *
Division: /
Modulus (Remainder): MOD

The following examples illustrate arithmetic operators in action.

Example:

```
1 + 1
2 - 1
3 * 4
100 / 10
25 MOD 3
10 MOD 2.5
11.6 MOD 3.1
```

Arithmetic operations can be grouped using parentheses. This ensures that specific operations are evaluated in the expected order, and can make your programs easier to understand.

Example:

```
# These all have different results
SET number a = 5 * 5 / 10 + 2
SET number b = 5 * 5 / (10 + 2)
SET number c = 5 * (5 / 10 + 2)
WriteLn(a, b, c)
```

Order of Operations

Arithmetic operators are evaluated in the following order:

1. Parentheses
2. Division/Modulo
3. Multiplication
4. Addition
5. Subtraction

Example:

```
# RESULT: a is 10
SET number a = 110-100/10*10
# RESULT: b is 109
SET number b = 110-100/(10*10)
# RESULT: c is 110
SET number c = 110-100/10 MOD 10
```

Remember: Use parentheses to group operations as needed.

Advanced Arithmetic Functions

Several built-in arithmetic functions are provided for advanced mathematical operations:

Random
Round
Ceil
Floor
Abs
Pow
Sqrt
Sin
Cos
Tan
Atan
Log

See *Appendix 5: Built-in Functions* for details about each of these functions.

Chapter 6: Comparisons and Conditionals

In the previous chapter, you learned about math operators, i.e. +, -, *, /, MOD.

Another type of operator, the comparison operator, allows us to compare two values to get a boolean result.

Comparison Operators

The following operators compare the left and right values. They evaluate to true or false depending on the outcome of the comparisons.

Example:

```
# Less than
10 LT 20

# Greater than
20 GT 10

# Less than or equal to
20 LTE 20

# Greater than or equal to
10 GTE 10

# Equal
10 EQ 10

# Not Equal
10 NEQ 20
```

Conditionals

Comparison operators are most useful when combined with **IF** statements. The combination of these two elements creates a *conditional* statement.

Example:

```
SET number a = 10
SET number b = 5
```

```
# If a greater than b, do something
IF (a GT b) THEN
      WriteLn("a more than b")
OK
```

What if a is less than b ?

The **ELSE** keyword handles the situation where a conditional does not evaluate to true.

Example:

```
SET number a = 10
SET number b = 50

# If a greater than b, do something
IF (a GT b) THEN
      WriteLn("a more than b")
ELSE
      # Otherwise do this.
      WriteLn("b more than a")
OK
```

Chapter 7: Prefix and Logical Operators

Prefix Operators

Some operators affect the value that follows them.

The **NOT** operator inverts anything that follows it, and evaluates to a boolean (true or false).

Example:

```
WriteLn(not false)  # True
WriteLn(not true)   # False

# They can be stacked infinitely
WriteLn(not not not true)  # False

# Can be used with variables
SET number a = 1
SET boolean b = not a   # False
```

The *negation* (**-**) operator inverts the sign of a number, making a positive number negative or a negative number positive.

Example:

```
SET number a = 123
SET number b = -a
WriteLn(a, b)

# Using with a number
SET number c = -345
SET number d = -c
```

Logical Operators

Logical operators compare two values and return the outcome as a boolean result of true or false.

Like comparison operators, they are useful when combined with **IF/THEN/ELSE**, as in the following examples.

AND

AND produces a true result when both of the compared values produce a true result.

Example:

```
# Test values
SET boolean a = true
SET boolean b = true

# a and b are true, therefore we
# get "true"
WriteLn(a AND b)

SET boolean c = false

# a is true, but c is false,
# therefore we get "false"
WriteLn(a AND c)
```

OR

OR produces a true result when either of the compared values produce a true result.

Example:

```
# Test values
SET boolean a = true
SET boolean b = false

# a is false, but b is true, so we
# get "true"
WriteLn(a OR b)
```

XOR

XOR is short for "Exclusive OR", and produces a true result when either of the compared values are true, but not when both values are true.

Example:

```
# Test values
SET boolean a = true
SET boolean b = false

# a is true, but b is false, so we
# get "true"
WriteLn(a XOR b)
```

```
SET boolean c = true

# a is true and c is true, so we get
# "false"
WriteLn(a XOR c)
```

NAND

NAND means "NOT AND", and produces a true result when either of the compared values is not true. If both values are true, then NAND produces false.

Example:

```
# Test values
SET boolean a = false
SET boolean b = false

# a and b are false, therefore we
# get "true"
WriteLn(a NAND b)

# Changing either value to true
# does not change the result.
a = true
WriteLn(a NAND b)

# Changing both values to true does.
b = true
WriteLn(a NAND b)
```

Think of NAND as the inverse of AND.

NOR

NOR means "NOT OR", and produces a true result when neither of the compared values is true. If either value is true, NOR produces false.

Example:

```
# Test values
SET boolean a = false
SET boolean b = false

# a and b are false, therefore we
```

```
# get "true"
WriteLn(a NOR b)

# Changing either value to true
# results in "false"
a = true
WriteLn(a NOR b)
```

Think of NOR as the inverse of OR.

XNOR

XNOR means "NOT Exclusive OR", and produces a true result when neither of the compared values are true, or when both values are true.

Example:

```
# Test values
SET boolean a = true
SET boolean b = false

# a is true, but b is false, so we
# get "false"
WriteLn(a XNOR b)

SET boolean c = true

# a is true and c is true, so we get
# "true".
WriteLn(a XNOR c)
```

Think of XNOR as the inverse of XOR.

Logical Operators and Types

Logical operators can be used with types other than boolean. When comparing non-boolean types, the following rules are used:

- When comparing a number, non-zero values are true, and 0 is false.
- When comparing a string, non-empty strings are true, and "" is false.
- When comparing a map, if the map has any keys set, it is true.
- When comparing a list, if the list contains any values, it is true.
- When comparing a defined function or built-in, the result is true.

Example:

```
# A non-zero number
WriteLn(1 AND true)

# A non-empty string
WriteLn("Hello" AND true)

# A map with keys
WriteLn({"name": "John"} AND true)

# A list with values
WriteLn([1, 2, 3] AND true)
```

Combining Logical Operators

Just like other operators, the logical operators can be combined.

Example:

```
IF (1 XOR 0)
      AND (0 XOR 1)
      AND (0 NAND 0)
      AND (0 OR 0 OR 1)
THEN
      WriteLn("It works!")
OK

WriteLn(0 OR 0 OR 0 OR 0 OR 1)
```

Chapter 8: Loops and Iterators

Looping

WHILE creates a loop that repeats until a condition is no longer met.

Example:

```
READY number counter
# Repeat as long as counter is less
# than 10.
WHILE counter LT 10 REPEAT
      WriteLn(counter)
      counter = counter + 1
DONE
```

The following WHILE loop creates a forever loop. It will continue to run until the HALT (F6) is pressed:

```
WHILE true REPEAT
      WriteLn("Loop!")
DONE
```

Iterating

EACH iterates through a list. It is similar to a loop, except that it will stop once all the values in the list have been iterated over.

Example:

```
SET list nums = [1, 2, 3]
EACH key, value IN nums REPEAT
      WriteLn(key, value)
DONE
```

When iterating, the key contains the position in the list (starting at zero), and the value contains the current value from the list.

Note: the *key* is optional, and may be omitted.

Example:

```
SET list nums = [1, 2, 3]
```

```
EACH value IN nums REPEAT
      WriteLn(value)
DONE
```

When *key* is omitted, it is not available inside the loop.

Note: "key" and "value" are arbitrary names chosen for these examples. Like any other variable, any valid name can be used.

Example:

```
SET list weekdays = [
      "Mo", "Tu", "We", "Th", "Fr"
]

EACH num, day IN weekdays REPEAT
      WriteLn(num, day)
DONE
```

EACH also supports inline lists, as well as lists returned from function calls.

Example:

```
# Using inline list
EACH num IN [1, 2, 3] REPEAT
      WriteLn(num)
DONE

# Using function call
FUNCTION Nums()
      OUTPUT [1, 2, 3]
RESULT list

EACH num IN Nums() REPEAT
      WriteLn(num)
DONE
```

Chapter 9: Functions

Functions provide reusable blocks of code that can be called throughout a program.

Let's look at some examples of functions in action.

Example:

```
# A function that displays a
# greeting to the user.
FUNCTION SayHello()
      WriteLn("Hello User!")
RESULT nothing
```

To call a Function: Example:

```
# Call the greeting function
SayHello()
```

Functions can accept values supplied to them. These are called parameters.

Example:

```
# A function that takes two values.
# The function will add and print
# them.
FUNCTION ShowSum(number a, number b)
      WriteLn(a + b)
RESULT nothing
```

To call a function using parameters, do:

```
# Call the show sum function.
ShowSum(10, 20)
```

Functions can also be written to output values when called.

Example:

```
# A function that outputs a
# value when called.
FUNCTION Sum(number a, number b)
      # Output the sum of numbers
      OUTPUT a + b
RESULT number
```

The **OUTPUT** keyword tells the function to return the result. The **RESULT** keyword at the end of the function defines the type of value that is returned.

A value **OUTPUT** from a function can be stored in a variable that matches the type specified by the **RESULT** type.

Example:

```
# Call the Sum function from earlier
# and store the output result in a
# variable.
SET number total = Sum(10, 20)

# Display the result
WriteLn(total)
```

When a function does not use OUTPUT, use the special type "nothing" to tell the evaluator that the function does not produce a result.

Example:

```
# This function does not output
# anything.
FUNCTION SayHello()
      WriteLn("Hello User!")
RESULT nothing

# Setting the result to a variable
# will cause an error.
SET string a = SayHello()
```

Built-in Functions

Built-in functions are predefined by *FREMScript* and can be used throughout programs just like user-defined functions. Write(), WriteLn(), and Cls() are examples of built-in functions.

See *Appendix 5: Built-in Functions* for a full reference of all built-in functions.

Chapter 10: The Pixel Layer

The Sprite32! pixel layer handles the rendering of individual pixels to the screen for X (0-319) and Y (0-199) coordinates.

To use the pixel layer, first ensure it is enabled using the following System Boolean Register flag:

```
# Enable the pixel layer
SysPut("B", 30, true)
```

To disable it later, set it to "false".

The following code will draw 256 red pixels to random locations on the screen.

Example:

```
SET number red = 255
SET number blue = 0
SET number green = 0
SET number alpha = 1

EACH y in Range(0, 256) REPEAT
      Pixel(
            Random(0, 319),
            Random(0, 199),
            red,
            green,
            blue,
            alpha
      )
DONE

DrawPixels()
```

To hide any text on the screen, either use the Cls() command, or set the following system command to disable text entirely.

Example:

```
# Disable the text layer
# Set to "true" to re-enable text.
SysPut("B", 10, false)
```

Clearing Pixels

Use the ClearPixels() command to clear any pixels drawn on the screen during the next system screen update.
You can also press the CTRL+F7 shortcut on the keyboard to clear all text and graphics layers, including the Pixel Layer.

When clearing pixels, it is important to give the system a moment to process the command before resuming the drawing of new pixels. Otherwise, you may find that the draw commands occur before the clear command has finished, resulting in some pixels seemingly being "erased".

Consider the following program which illustrates the problem.

Example:

```
# Enable the pixel layer
SysPut("B", 30, true)

# Disable the text layer
SysPut("B", 10, false)

ClearPixels()

EACH y in Range(0, 4) REPEAT
      Pixel(
            Random(0, 319),
            Random(0, 199),
            255,  # Red
            0,    # Green
            0,    # Blue
            1     # Alpha
      )
DONE

DrawPixels()
```

Running the above program several times should illustrate how ClearPixels() and DrawPixels() overlap. Adding a bit of delay gives the system enough time to finish clearing before continuing with draws.

Example:

```
# Clear Pixels
```

```
ClearPixels()

# Wait a moment
Wait(.1)

# Draw pixel
Pixel(0, 0, 255, 0, 0, 1)

# Render all pixels
DrawPixels()
```

Using Sync()

The Sync command is built to address the timing issues that occur when clearing and drawing on the Pixel, Grid, and Sprite layers. It instructs the system to draw all three layers, then Wait for 100 ms to allow the system to finish processing.

Using the example from the previous section, replace the Wait() command with Sync().

Example:

```
# Clear Pixels
ClearPixels()

# Synchronize screen draws
Sync()

# Draw pixel
Pixel(0, 0, 255, 0, 0, 1)

# Render all pixels
DrawPixels()
```

When clearing the Pixel Layer, it is reinitialized, meaning any pixels previously plotted with the Pixel() function are lost. If you only need to hide the Pixel layer temporarily, use the system disable flag instead: *`SysPut("B", 30, false)`*

Performance

Drawing individual pixels is an expensive process. Here are some tips for maximizing speed when drawing on the pixel layer:

- Draw only what you need, utilize sprites and grid to handle draws across larger areas.
- Call DrawPixels() only when necessary.

- If drawing pixels in a loop, try to call DrawPixels() at the end, rather than after every single draw.

The following program uses math functions to draw a sine wave. Observe where *DrawPixels()* is executed in order to balance performance and visual feedback:

```
SysPut("B", 30, true)
Cls()
ClearPixels()
Sync()

# Draw a sine wave
EACH x in Range(0, 319) REPEAT
      SET number y =
            100 + Sin((x/2)/30)*90
      Pixel(x, ToInt(y), 255, 0, 0, 1)
DONE

DrawPixels()
```

Draw a square:

```
SysPut("B", 30, true)
Cls()
ClearPixels()
Sync()

# Draw top, bottom
EACH x in Range(20, 120) REPEAT
      EACH y in [20, 120] REPEAT
            Pixel(x, y, 255, 0, 0, 1)
      DONE
DONE

# Draw sides
EACH x in [20, 120] REPEAT
      EACH y in Range(20, 120) REPEAT
            Pixel(x, y, 255, 0, 0, 1)
      DONE
DONE

DrawPixels()
```

Chapter 11: The Grid Layer

The Sprite32! grid layer displays 8x8 solid color squares for X (0-39) and Y (0-24) coordinates. A total of 1000 squares can be displayed using the grid system.

To use the grid layer, first ensure it is enabled using the following System Boolean Register flag:

```
# Enable the grid layer
SysPut("B", 20, true)
```

To disable it later, set it to "false".

Once the grid is enabled, you can begin drawing to it.

Example:

```
# Draw color Grid squares on screen.
SET list xCoords = Range(0, 39)
SET list yCoords = Range(0, 24)

EACH y IN yCoords REPEAT
      EACH x IN xCoords REPEAT
            SET number red
                  = Random(0,255)
            SET number green
                  = Random(0, 255)
            SET number blue
                  = Random(0, 255)

            Grid(
                  x, y,
                  red, green, blue, 1
            )
      DONE
DONE
```

To display the squares the have been plotted, tell the system to render the grid layer:

```
# Draw pending grid changes
DrawGrid()
```

To hide any text on the screen, either use the *Cls()* command, or set the following system command to disable text entirely.

Example:

```
# Disable the text layer
# Set to "true" to re-enable text.
SysPut("B", 10, false)
```

Clearing Grid Squares

Use the ClearGrid() command to clear any squares drawn on the screen during the next system screen update.

You can also press the **CTRL+F7** shortcut on the keyboard to clear all text and graphics layers, including the Grid Layer.

The same timing problem described for Pixels in Chapter 10 exists for Grid. Be sure to use a small amount of delay after clearing the Grid layer prior to drawing new Grid points to prevent synchronization issues.

Example:

```
# Clear Grid, then delay.
ClearGrid()
Wait(.1)

# Optionally use Sync() instead of
# Wait()
ClearGrid()
Sync()
```

When clearing the Grid Layer, it is reinitialized, meaning any squares previously plotted with the Grid() function are lost. If you only need to hide the Grid layer temporarily, use the system disable flag instead: *`SysPut("B", 20, false)`*

Chapter 12: The Sprite Layer

The Sprite32! sprite layer displays 8x8 user-defined sprites at X (0-319) and Y (0-199) coordinates. A total of 32 sprites are available for display, hence the name of the computer!

Sprites can be moved about the screen, scaled up and down, and flipped in various orientations.

To use the sprite layer, first ensure it is enabled using the following System Boolean Register flag.

Example:

```
# Enable the sprite layer
SysPut("B", 40, true)
```

To disable it later, set it to false.

To enable individual sprites (0-31) for display and manipulation, use the system flag for each sprite, beginning at 42.

```
# Enable Sprite 0
SysPut("B", 42, true)

...

# Enable Sprite 31
SysPut("B", 73, true)
```

See Appendix 2: System Boolean Registers for a full list of sprite flags.

Creating and Drawing Sprites

Sprites are defined as 8x8 pixel grids, for a total of 64 pixels per sprite. Each sprite pixel is drawn in the chosen color using the SpriteSet() function.

Example:

```
Cls()
# Enable the sprite layer
SysPut("B", 40, true)

# Enable sprite 0
```

```
SysPut("B", 42, true)

# Draw the first pixel in top left
# corner of the sprite in bright red
SpriteSet(0, 0, 0, 255, 0, 0, 1)

# Place the sprite in top left
# corner of screen.
SpritePos(0, 0, 0)

# Draw the sprite
DrawSprites()
```

When running the example, you should see a single red pixel. To more clearly illustrate the sprite boundaries, modify the example with the following loops.

Example:

```
Cls()
# Enable the sprite layer
SysPut("B", 40, true)

# Enable sprite 0
SysPut("B", 42, true)

# Fill the entire sprite with gray
# color.
EACH y in Range(0, 7) REPEAT
      EACH x in Range(0, 7) REPEAT
            SpriteSet(
                  0, x, y,
                  128, 128, 128, 1
            )
      DONE
DONE

# Draw the first pixel in top left
# corner of the sprite in bright red
SpriteSet(0, 0, 0, 255, 0, 0, 1)

# Place the sprite in top left
# corner of screen.
SpritePos(0, 0, 0)
```

```
# Draw the sprite
DrawSprites()
```

Now, the sprite should have a gray background with a single red pixel. Try moving the pixel to bottom right corner by changing the SpriteSet line:

```
# Draw green pixel at bottom right
SpriteSet(0, 7, 7, 0, 255, 0, 1)
```

Moving Sprites

Once a sprite has been created, it can be moved around the screen. In the previous examples, we placed the sprite at 0, 0 (top left corner).

Continuing from the above examples, move the sprite to the bottom right corner by modifying the SpritePos line:

```
SpritePos(0, 311, 191)
```

Place it in the middle of the screen:

```
SpritePos(0, 152, 92)
```

Note: Created sprites are preserved in system memory until the next RESET.

Sprite Boundaries

Sprites can be placed at pixel-precise locations on the screen, and even moved "off-screen". The supported move ranges for sprites are:

-64 to 320 on the x-axis
-64 to 200 on the y-axis

Scaling Sprites

Sprites can be scaled on the x and y axes using SpriteScale() function. Sprites can be scaled 1 to 8 times their original size using scale (0-7) on either the x or y-axis. Use the same value on both axes to scale uniformly, or "stretch" the sprite by using different values on one axis.

Example:

```
# Scale sprite to twice the default
```

```
# size.
SpriteScale(0, 1, 1)

# 4x
SpriteScale(0, 3, 3)

# 8x
SpriteScale(0, 7, 7)

# X-stretch multiplier of 8
SpriteScale(0, 7, 0)

# Reset scale
SpriteScale(0, 0, 0)
```

Orienting Sprites

Sprites can be flipped on their x and y axes using the SpriteFlip() function.

Example:

```
# Mirror sprite on the y-axis
SpriteFlip(0, 1, -1)

# Mirror sprite on the x-axis
SpriteFlip(0, -1, 1)

# Mirror sprite on both axes
SpriteFlip(0, -1, -1)
```

Clearing Sprites

Use the ClearSprites() command to clear any sprites drawn on the screen during the next system screen update. You can also press the **CTRL+F7** shortcut on the keyboard to clear all text and graphics layers, including the Sprite Layer.

When clearing sprites, it is important to give the system a moment to process the command before resuming the drawing of new sprites. Otherwise, you may find that the draw commands occur before the clear command has finished, resulting in some sprites seemingly being "erased".

Consider the following program which illustrates the problem:

```
Cls()
```

```
# Enable the sprite layer
SysPut("B", 40, true)

# Enable sprite 0
SysPut("B", 42, true)

# Disable the text layer
SysPut("B", 10, false)

ClearSprites()

EACH y in Range(0, 4) REPEAT
      # Fill the entire sprite with gray
      # color.
    EACH y in Range(0, 7) REPEAT
      EACH x in Range(0, 7) REPEAT
            SpriteSet(
                  0, x, y,
                  128, 128, 128, 1
            )
      DONE
    DONE

      SpriteSet(0, 0, 0, 255, 0, 0, 1)
    SpritePos(0, 0, 0)
    DrawSprites()
DONE

DrawSprites()
```

Running the above program several times should illustrate how ClearSprites() and DrawSprites() overlap. Adding a bit of delay gives the system enough time to finish clearing before continuing with draws.

Example:

```
# Clear sprites
ClearSprites()

# Wait a moment
Wait(.1)

EACH y in Range(0, 4) REPEAT
      # Fill the entire sprite with gray
```

```
    # color.
    EACH y in Range(0, 7) REPEAT
      EACH x in Range(0, 7) REPEAT
            SpriteSet(
                  0, x, y,
                  128, 128, 128, 1
            )
      DONE
    DONE

      SpriteSet(0, 0, 0, 255, 0, 0, 1)
    SpritePos(0, 0, 0)
DONE

# Render all sprites
DrawSprites()
```

Using Sync()

The Sync command is built to address the timing issues that occur when clearing and drawing on the Pixel, Grid, and Sprite layers. It instructs the system to draw all three layers, then Wait for 100 ms to allow the system to finish processing.

Using the example from the previous section, replace the *Wait()* command with *Sync()*.

Example:

```
# Clear sprites
ClearSprites()

# Synchronize screen draws
Sync()

EACH y in Range(0, 4) REPEAT
      # Fill the entire sprite with gray
      # color.
      EACH y in Range(0, 7) REPEAT
      EACH x in Range(0, 7) REPEAT
            SpriteSet(
                  0, x, y,
                  128, 128, 128, 1
            )
      DONE
```

```
    DONE

      SpriteSet(0, 0, 0, 255, 0, 0, 1)
      SpritePos(0, 0, 0)
DONE

# Render all sprites
DrawSprites()
```

When clearing the Sprite Layer, it is reinitialized, meaning any sprites previously plotted with the SpriteSet() function are lost. If you only need to hide the Sprite layer temporarily, use the system disable flag instead: *`SysPut("B", 40, false)`*

Or disable the specific sprite that you'd like to hide:

```
# Hide sprite 0
SysPut("B", 42, false)
```

Chapter 13: Audio

The Sprite32! provides 3 channels of audio generation using **ADSR** (Attack, Decay, Sustain, Release) envelopes. Each audio channel supports five distinct waveforms, and 12 notes over 10 octaves.

Tones are played using the Sprite32! audio coprocessor, which plays all 3 channels of audio in parallel during evaluation.

Note: *The Sprite32!* hardware is not particularly suited to deal with the complexities of precise audio timing. In conjunction with the relative speed of *FREMScript* evaluation, composing complex pieces of music using consistent synchronization of layers can prove difficult.

Tip: If you are currently listening to the soundtrack audio, press **Menu (F2)** to view options for muting the soundtrack while testing audio!

Example:

```
# Play a basic square wave note of
# C2 for 1 second.
SET number channel = 0
SET number shape = 2
SET number octave = 2
SET number note = 0
SET number attack = 0
SET number decay = 0
SET number sustain = 1
SET number release = 1

# Set the channel, shape, tone.
Tone(
      channel,
      shape,
      octave,
      note,
      attack,
      decay,
      sustain,
      release
)

# Open the gate to play the note.
Gate(0, true)
```

```
# As long as the gate remains open,
# the audio will hold at the sustain
# stage. Increasing the delay here
# will cause the note to play for
# longer.
Wait(1)

# Close the gate to end the note.
# The release stage is invoked when
# the gate is closed.
Gate(0, false)

# Audio buffers clear at end of
# program execution. Wat a moment to
# let the sound finish the release
# length we set (1s)
Wait(1)
```

Channels

There are three channels available. The channel number (0-2) is supplied as the first argument to the Tone() function.

Channel volume is in decibels (dB), and can range from -80 (inaudible) to 0 (unity). The default volume for all channels is -12dB. This allows for a nice balance between all three channels when they are playing simultaneously. Use the Volume() function to adjust the audio levels for each channel.

Example:

```
# Turn up the volume on channel 1
Volume(0, -3)
```

Waveforms

The following audio waveforms are available, and are supplied as the second argument to the Tone()
function:

- 0: Sine
- 1: Triangle
- 2: Square
- 3: Saw

- 4: Noise

Octaves

Ten octaves are available for each waveform. Octaves (0-9) are supplied as the third argument to the Tone() function.

Notes

The following frequencies represent the available notes starting at C0. Notes (0-11) are supplied as the fourth argument to the Tone() function.

Equal-tempered scale frequencies in Hz.

A4 = 440 Hz

Note	**Frequency in Hz**
C	16.35
C#	17.32
D	18.35
Eb	19.45
E	20.60
F	21.83
F#	23.12
G	24.50
G#	25.96
A	27.50
Bb	29.14
B	30.87

ADSR Envelope

The Attack, Decay, Sustain, and Release stages of each generated tone can be controlled to produce various sound effects. The following subsections give a quick overview of each stage.

Attack

Attack is the number of seconds it takes the tone to go from off to full volume. It is constrained by the volume of the channel, set using the Volume() function.

Decay

After the Attack stage has concluded, Decay is the number of seconds it takes the tone to go from full volume to the value defined for Sustain. It is constrained by the volume of the channel, set using the Volume() function.

Sustain

At the end of the Decay stage, Sustain represents the volume at which the tone is held as long as the gate
remains open. It is represented as a ratio of the channel volume, ranging from 0.0 to 1.0.

Note: that when Sustain is set to 1.0, the effect of the Decay stage will not be noticed.

Release

Release is the number of seconds it takes the note to go from Sustain volume to off after the gate is closed.

Note: It is important to give the system time to finish playing audio during the release stage. This can be accomplished by providing delays using the *Wait()* function. Otherwise, the audio will be cut off and can result in popping.

Looping Audio

The following code will play a tone in loop:

```
SET number channel = 0
SET number shape = 3
SET number octave = 3
SET number note = 0
SET number attack = .5
SET number decay = .5
SET number sustain = .95
SET number release = .5
Tone(
      channel,
```

```
      shape,
      octave,
      note,
      attack,
      decay,
      sustain,
      release
)

WHILE true REPEAT
      Gate(0, true)
      Wait(1)
      Gate(0, false)
      Wait(1)
DONE
```

Play percussive noise in loop:

```
SET number channel = 0
SET number shape = 4
SET number octave = 0
SET number note = 0
SET number attack = 0
SET number decay = 0
SET number sustain = 1
SET number release = 0

Tone(
      channel,
      shape,
      octave,
      note,
      attack,
      decay,
      sustain,
      release
)

WHILE true REPEAT
      Gate(0, true)
      Wait(.2)
      Gate(0, false)
      Wait(.1)
```

DONE

Chapter 14: Data Files

Data can be created and accessed from *FREMScript* using built-in functions.

Note: Data files are stored on the system disk with the .dat extension. (Press F2 to open the menu, then "Open Directory" to view the system disk)

DataWrite

Write a string to *data file* using (A)ppend or (W)rite modes.

Append

Using mode "A" appends data to new line of an existing file. An evaluation error occurs if the file does not exist.

Write

Using mode "W" writes data to a file. If the file does not exist, it is created. If a file already exists, it is overwritten.

Example:

```
# Create/overwrite data in test.dat
DataWrite("test", "W", "Hello")

# Append to data in test.dat
DataWrite("test", "A", "World")
```

Data files are stored on the system disk with the .dat extension. The extension is inferred when calling Data write/read functions, do not specify it.

DataRead

Read all lines from data file as a list.

Example:

```
# Read all lines from data.
SET list lines = DataRead("test")
EACH line IN lines REPEAT
      WriteLn(line)
DONE
```

DataReadLine

Read single line from data file at position.

Position range is 0 to total number of lines - 1. If the specified line does not exist, (nothing) is returned.

Example:

```
# Read the first line from data.
SET string line = DataReadLine("test", 0)
WriteLn(line)
```

Chapter 15: Input

User input can be captured from the host operating system's mouse and keyboard.

Working with Keyboard Input

Keyboard input can be captured using the KeyDown() function. The KeyDown() function outputs the keyboard key that is currently being pressed. When no key is being pressed, "" is output.

Note: KeyDown() will output the first pressed key it finds. When multiple keys are held, the first key in the internal list will be output.

Example:

```
WHILE true REPEAT
      WriteLn(KeyDown())
DONE
```

Connecting keyboard input

Keyboard input can be relayed to the system instead of the host interface by using the System Boolean Register.

Example:

```
# Connect host keyboard to hardware.
SysPut("B", 5, true)

WHILE true REPEAT
      WriteLn(KeyDown())
DONE
```

While the program is running, all supported keystrokes will be relayed to the system. Function keys maintain their behavior on the host system. To disconnect the input, press F6 to halt the program, or set the register to false.

Working with Mouse Input

The mouse position can be detected using the MousePos() function. The MousePos() function outputs a map of the last recorded X and Y screen coordinates of the mouse position. Mouse position is only detected within the 320x200 screen area excluding the screen border.

Note: MousePos() outputs screen coordinates between 0-319x and 0-199y

Example:

```
# Color the screen to show boundaries
SysPut("N", 21, 0)

# Display mouse coordinates on screen
WHILE true REPEAT
      Dump(MousePos())
DONE
```

Detecting mouse button presses

The primary/secondary mouse button presses can be detected using the MouseButton() function. MouseButton() outputs boolean true when the specified button is pressed, and false otherwise. Mouse button states are only detected within the 320x200 screen area excluding the screen border.

Button Number	Physical Button
0	Primary button (left click)
1	Secondary button (right click)

Example:

```
WHILE true REPEAT
      IF MouseButton(0) EQ true THEN
            Dump("LEFT BUTTON")
            Wait(.1)
      OK

      IF MouseButton(1) EQ true THEN
            Dump("RIGHT BUTTON")
            Wait(.1)
      OK
      Cls()
DONE
```

Chapter 16: Registers & Function Cache

Examples throughout this manual use SysPut() and SysGet() calls to write to and read from the three system registers: *Boolean, Number, and String*. These registers control the behavior of the Sprite32! System, and are preserved between program runs until the computer is powered off.

Boolean Register

The Boolean Register stores true/false flags which control various system states. See *Appendix 2: System Boolean Registers* for a full list of Boolean Register settings.

Example:

```
# Capture input from host keyboard
# by setting Boolean Register 5
SysPut("B", 5, true)

# Enable the Sprite Layer by setting
# Boolean Register 40
SysPut("B", 40, true)

# Enable the Text layer.
SysPut("B", 10, true)
```

Values can be read from the boolean register using SysGet().

Example:

```
# Check whether Sprite 0 is
# enabled (true) or disabled (false)
SET boolean en = SysGet("B", 42)
WriteLn(en)
```

Number Register

The Number Register stores numeric values used by various system processes. See *Appendix 3: System Number Registers* for a full list of Number Register settings.

Example:

```
# Set the maximum number of lines to
# be displayed on the screen (0-25)
SysPut("N", 30, 10)
```

```
# Set the current screen color
# palette.
SysPut("N", 20, 4)

# Set the background color to color
# index from color palette.
SysPut("N", 21, 7)

# Border color
SysPut("N", 22, 9)

# Font color
SysPut("N", 23, 3)
```

Values can be read from the number register using SysGet().

Example:

```
# Get the UNIX timestamp of system
# power on.
SET number ts = SysGet("N", 1)

# Get the seed used by Random()
# set during power/reset cycle.
SET number seed = SysGet("N", 50)

Dump(ts, seed)
```

String Register

The String Register stores text values used in various system messages. Values can be read from the string register using SysGet(). See *Appendix 4: System String Registers* for a full list of String Register settings.

Example:

```
# Get the version/build number
SET string ver = SysGet("S", 8)
WriteLn(ver)
```

User Registers

In addition to the System Registers, three User Registers are available for storing information between program runs: Boolean, Number, String. Each register type performs the same function and contains the same number of positions as the System Registers. User Registers are preserved in memory until the computer is powered off.

Note: User Registers are designed for you to use in your programs, and do not have any direct impact on the system. By default, they are empty at power up.

Example:

```
# Program A sets a value to the user
# register.
MemPut("N", 0, 123)
```

Enter the above program, then press CTRL+N to start a new one. Enter the following program to view the value stored in user memory register.

Example:

```
# Program B reads the number
SET number n = MemGet("N", 0)
WriteLn(n)
```

Function Cache

Similar to System/User register memory, the Function Cache stores function declarations in memory for later use. Use CachePut(index, functionName) to store a function in the cache, and CacheGet(index) to retrieve it.

A total of 256 functions may be stored in cache positions 0-255. Any functions stored in cache consume a portion of the program memory available to the system.

Just like registers, the Function Cache is preserved in memory until the computer is powered off.

Example:

```
# Declare a function that sums two
# numbers.
FUNCTION sum (number a, number b)
      OUTPUT a + b
RESULT number
```

```
# Store the function in cache 0
CachePut(0, sum)
```

Enter the above program, then press CTRL+N to start a new one. Enter the program below to load the function from cache and call it.

Example:

```
SET function sum = CacheGet(0)
WriteLn(sum(9, 9))
```

Note: you can also call the Function Cache directly:

```
# Sum two numbers using function.
WriteLn(CacheGet(0)(12, 12))
```

Chapter 17: Includes & Defines

You can reuse code you've written in one solution in another by using the **INCLUDE** statement to load *FREMScript* from a file saved in the user directory.

Suppose you had a file hello.frem with the following code:

```
SET string greeting = "Hello!"
```

In the code editor, you can import that code like so:

```
INCLUDE "hello"
WriteLn(greeting)
```

When importing code in this way, any errors that occur will reference the file name where the error occurred in the status window output.

Note: INCLUDE does not raise any errors when the specified file or path cannot be found.

Defining and using Macros

The **DEFINE** search replacement statement instructs the parser to replace all occurrences of search with replacement in the remaining lines of code.

Example:

```
DEFINE PROJECT_DIR "username/files"
WriteLn(PROJECT_DIR)
```

After parsing, the code the evaluator receives will be:

```
# Blank line
WriteLn("username/files")
```

The first string literal following the DEFINE keyword is the search, and the remainder of the line following the next blank space is the replacement.

When applying DEFINE statements, the parser discards the statements once the replacements have been applied, so there is no further impact during evaluation.

Note: Replacements affect all subsequent lines, even other DEFINE statements. Consider the following example where the first DEFINE affects any following lines, breaking the intended outcome.

Example:

```
DEFINE DIR "my_project"

# Next line is now: DEFINE "my_project"ECTION "up"
DEFINE DIRECTION "up"
```

Additionally, DEFINE statements cannot use the search in the replacement. When invalid DEFINE statements are encountered, they are ignored by the Parser.

Example:

```
# Invalid DEFINE is ignored
DEFINE MESSAGE MESSAGE

# MESSAGE not defined
WriteLn("MESSAGE")

# Valid DEFINE is applied
DEFINE MESSAGE HELLO!

# MESSAGE  defined
WriteLn("MESSAGE")
```

Be careful when using DEFINE statements. They are powerful, but can result in unexpected behavior!

Appendix 1: Color Palettes

The following color palettes are available for use:

(0) FREM Standard Palette (Default)
(1) C64 NTSC
(2) C64 PAL
(3) C64 TRUE COLOR
(4) CGA16
(5) GRAYSCALE
(6) - *EMPTY* -
(7) - *EMPTY* -
(8) - *EMPTY* -
(9) USER

Set desired color palette in the System Number Register using the command:

```
SysPut("N", 20, palette_number)
```

Each palette consists of 16 24-bit RGB colors.

(0) FREM Standard Palette (Default)

Color Number	Color Name	Color RGB Values
0	Black	(0, 0, 0)
1	Dark Gray	(32, 32, 32)
2	Medium Gray	(80, 80, 80)
3	Light Gray	(128, 128, 128)
4	Red[	(240, 32, 32)
5	Orange	(240, 128, 0)
6	Yellow	(240, 240, 64)
7	Pink	(255, 192, 192)
8	Green	(32, 240, 32)
9	Light Green	(192, 240, 192)
10	Light Blue	(188, 235, 221)
11	Blue	(32, 32, 240)
12	Purple	(255, 0, 255)
13	Lavender	(243, 213, 245)
14	Almost White	(224, 224, 224)

15	White	(255, 255, 255)

(9) USER Palette

Use this color palette to define custom user colors. See *Appendix 3: System Number Registers* for user color registers 64-111.

Appendix 2: System Boolean Registers

Index	Flag	Function
0	HALT	Halt program execution
1	OUT OF MEMORY	User program memory exhausted
2	RESET	Signal to reset system
3	POWER OFF	Signal to power off system
4	IGNORE HARDWARE INPUT	Ignore any input from the hardware interface
5	LOCK KEYBOARD TO HARDWARE	Lock the Host keyboard to the hardware interface. When set in the program, the host machine keyboard will be locked and a dialog will be displayed over the editor. After the program has completed, this flag is reset to False.
9	SCREEN ENABLED	Screen display enabled
10	TEXT LAYER ENABLED	Text display layer enabled
11	TEXT UPPERCASE	Draws all screen text in uppercase characters only
20	GRID LAYER ENABLED	Grid display layer enabled
21	INITIALIZE GRID	Signal to reset Grid to initial state. After initialization, this flag is set to False.
30	PIXEL LAYER ENABLED	Pixel layer enabled
31	INITIALIZE PIXELS	Signal to clear all pixel draws from the Pixel layer. After initialization, this flag is set to False.
40	SPRITE LAYER ENABLED	Sprite display layer enabled
41	INITIALIZE SPRITES	Signal to reset Sprites to initial state. After initialization, this flag is set to False.
42	SPRITE 0 ENABLED	Enable Sprite 0
43	SPRITE 1 ENABLED	Enable Sprite 1
44	SPRITE 2 ENABLED	Enable Sprite 2
45	SPRITE 3 ENABLED	Enable Sprite 3
46	SPRITE 4 ENABLED	Enable Sprite 4
47	SPRITE 5 ENABLED	Enable Sprite 5
48	SPRITE 6 ENABLED	Enable Sprite 6
49	SPRITE 7 ENABLED	Enable Sprite 7
50	SPRITE 8 ENABLED	Enable Sprite 8

51	SPRITE 9 ENABLED	Enable Sprite 9
52	SPRITE 10 ENABLED	Enable Sprite 10
53	SPRITE 11 ENABLED	Enable Sprite 11
54	SPRITE 12 ENABLED	Enable Sprite 12
55	SPRITE 13 ENABLED	Enable Sprite 13
56	SPRITE 14 ENABLED	Enable Sprite 14
57	SPRITE 15 ENABLED	Enable Sprite 15
58	SPRITE 16 ENABLED	Enable Sprite 16
59	SPRITE 17 ENABLED	Enable Sprite 17
60	SPRITE 18 ENABLED	Enable Sprite 18
61	SPRITE 19 ENABLED	Enable Sprite 19
62	SPRITE 20 ENABLED	Enable Sprite 20
63	SPRITE 21 ENABLED	Enable Sprite 21
64	SPRITE 22 ENABLED	Enable Sprite 22
65	SPRITE 23 ENABLED	Enable Sprite 23
66	SPRITE 24 ENABLED	Enable Sprite 24
67	SPRITE 25 ENABLED	Enable Sprite 25
68	SPRITE 26 ENABLED	Enable Sprite 26
69	SPRITE 27 ENABLED	Enable Sprite 27
70	SPRITE 28 ENABLED	Enable Sprite 28
71	SPRITE 29 ENABLED	Enable Sprite 29
72	SPRITE 30 ENABLED	Enable Sprite 30
73	SPRITE 31 ENABLED	Enable Sprite 31

Appendix 3: System Number Registers

Index	Flag	Function
0	SYSTEM INSTALL TIME	The UNIX timestamp corresponding to the first test boot during factory installation.
1	START TIME	On power on, the current UNIX timestamp is set to this value
5	USER REGISTER MEMORY	The total amount of memory available in user registers (read only)
6	SYSTEM REGISTER MEMORY	The total amount of memory available in system registers (read only)
7	PROGRAM MEMORY MAX	The total amount of memory available to programs including evaluated code. This value is modifiable but limited to total RAM installed.
8	PROGRAM MEMORY USED	The total amount of program memory in use
20	COLOR PALETTE	The assigned color palette. Each color palette contains 16 colors.
21	BACKGROUND COLOR	The screen background color
22	BORDER COLOR	The screen border color
23	FONT COLOR	The screen text color
30	FONT MAX LINES	Max number of lines for display
31	FONT SPACING TOP	Number of empty pixels to draw above each line
32	FONT SPACING BOTTOM	Number of empty pixels to draw below each line
33	FONT VERTICAL ADJUST	Number of pixels to offset screen text by. Positive numbers move down, negative up. Constrained by limits of screen border.
34	FONT HORIZONTAL ADJUST	Number of pixels to offset screen text by. Positive numbers move right, negative left. Constrained by limits of screen border.
50	RANDOM SEED	Integer seed for random number generation, used by builtins like Random().
64	USER COLOR PALETTE	User Color 0 (Red)
65	USER COLOR PALETTE	User Color 0 (Green)
66	USER COLOR PALETTE	User Color 0 (Blue)
67	USER COLOR PALETTE	User Color 1 (Red)
68	USER COLOR PALETTE	User Color 1 (Green)
69	USER COLOR PALETTE	User Color 1 (Blue)
70	USER COLOR PALETTE	User Color 2 (Red)
71	USER COLOR PALETTE	User Color 2 (Green)

72	USER COLOR PALETTE	User Color 2 (Blue)
73	USER COLOR PALETTE	User Color 3 (Red)
74	USER COLOR PALETTE	User Color 3 (Green)
75	USER COLOR PALETTE	User Color 3 (Blue)
76	USER COLOR PALETTE	User Color 4 (Red)
77	USER COLOR PALETTE	User Color 4 (Green)
78	USER COLOR PALETTE	User Color 4 (Blue)
79	USER COLOR PALETTE	User Color 5 (Red)
80	USER COLOR PALETTE	User Color 5 (Green)
81	USER COLOR PALETTE	User Color 5 (Blue)
82	USER COLOR PALETTE	User Color 6 (Red)
83	USER COLOR PALETTE	User Color 6 (Green)
84	USER COLOR PALETTE	User Color 6 (Blue)
85	USER COLOR PALETTE	User Color 7 (Red)
86	USER COLOR PALETTE	User Color 7 (Green)
87	USER COLOR PALETTE	User Color 7 (Blue)
88	USER COLOR PALETTE	User Color 8 (Red)
89	USER COLOR PALETTE	User Color 8 (Green)
90	USER COLOR PALETTE	User Color 8 (Blue)
91	USER COLOR PALETTE	User Color 9 (Red)
92	USER COLOR PALETTE	User Color 9 (Green)
93	USER COLOR PALETTE	User Color 9 (Blue)
94	USER COLOR PALETTE	User Color 10 (Red)
95	USER COLOR PALETTE	User Color 10 (Green)
96	USER COLOR PALETTE	User Color 10 (Blue)
97	USER COLOR PALETTE	User Color 11 (Red)
98	USER COLOR PALETTE	User Color 11 (Green)
99	USER COLOR PALETTE	User Color 11 (Blue)
100	USER COLOR PALETTE	User Color 12 (Red)
101	USER COLOR PALETTE	User Color 12 (Green)
102	USER COLOR PALETTE	User Color 12 (Blue)
103	USER COLOR PALETTE	User Color 13 (Red)
104	USER COLOR PALETTE	User Color 13 (Green)
105	USER COLOR PALETTE	User Color 13 (Blue)
106	USER COLOR PALETTE	User Color 14 (Red)
107	USER COLOR PALETTE	User Color 14 (Green)

108	USER COLOR PALETTE	User Color 14 (Blue)
109	USER COLOR PALETTE	User Color 15 (Red)
110	USER COLOR PALETTE	User Color 15 (Green)
111	USER COLOR PALETTE	User Color 15 (Blue)

Appendix 4: System String Registers

Index	Flag
0	==
1	FREM Sprite32!
2	First Rate Embedded Microsystems
3	COPYRIGHT 1985
4	USER REGISTER MEMORY
5	SYSTEM REGISTER MEMORY
6	PROGRAM MEMORY
7	BYTES
8	Ver 1.0.4.0
9	Negative(-11) Games Division
10	Press RUN (F5) to execute program.
11	Press HALT (F6) to halt program.
12	Press CLEAR (F7) to clear screen text.
13	Press RESET (F9) to restart computer.
14	Press POWER (F10) to power on/off.
15	READY
16	Booting...
32	`!@#$%^&*()+,-./:;<=>?'"[\]_{|}~□
33	¡¢£¤¥¦§¨©ª«¬®¯°±²³´µ¶·¸¹º»¼½¾¿
34	0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ
35	abcdefghijklmnopqrstuvwxyz
36	ÀÁÂÃÄÅÆÇÈÉÊËÌÍÎÏÐÑÒÓÔÕÖ×ØÙÚÛÜÝÞß
37	àáâãäåæçèéêëìíîïðñòóôõö÷øùúûüýþÿ
38	ĀāĂăĄąĆćĈĉĊċČčĎďĐđĒēĔĕĖėĘęĚěĜĝĞğ
39	ĠġĢģĤĥĦħĨĩĪīĬĭĮįıIJijĴĵĶķĸĹĺĻļĽľĿ
40	ŀŁłŃńŅņŇňŉŊŋÕōŎŏŐőŒœŔŕŖŗŘřŚśŜŝŞş
41	ŠšŢťŦŧŨũŪūŬŭŮůŰűŲųŴŵŶŷŸŹźŻżŽžſƒȚț
42	’ˆˇ˘˙˚¸˛˜˝;΄΅Α΄Ε΄Η΄Ι΄Ο΄Υ΄Ω
43	ΐΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩΪΫάέήί
44	ϋαβγδεζηθικλμνξοπρςστυφχψωϊϋόύώ
45	ÈËЂЃЄЅІЇЈЉЊЋЌЍЎЏАБВГДЕЖЗИЙКЛМНОП

46	РСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмноп
47	рстуфхцчшщъыьэюяѐёђѓєsiїjљњћќйўџ
48	ЂЪЋѪѬѲѳѶѵҐґҒғҒғҖҗҘҙҚқҜҝҠҡҢңҤҥҪҫҬҭҮү
49	¥ұҶҷҸҹҺһІЖжҶҷІĂăÄäÆæĔĕƏəЖжӠӡ
50	ЙйЙйӦӧӨөӮӯӲӱӰӱҶҷҬҭЫыQqWw
51	–—‘’‚“”„†‡•…‰‹›⁄₳€₹₫₴₸₽
52	№™←↑→↓∂∆∏∑√∞∫≈≠≤≥▲►▼◄◊★☆♠♣♥♦ ♪fifl
53	oov
54	h
55	df a
56	u bw
149	END

Appendix 5: Built-in Functions

This Appendix contains a full listing of built-in functions organized into sections according to their purpose.

5.1 - VARIABLE & ASSIGNMENT FUNCTIONS

EnvGet

EnvGet(string)

Read data from environment. This is equivalent to evaluating a variable.

Example:

```
SET string a = "A"
Dump(EnvGet("a"))
```

EnvPut

EnvPut(string, value)

Store data in global environment. This is equivalent to using SET to declare a variable, except it is always written to the global environment. Normally, a variable declared inside a function is limited to the scope of that function, unless it was previously declared outside the function. With EnvPut, the value will always be treated as if it was declared outside the function. This is useful for declaring values that can be used in other functions or called directly from the global scope.

Example:

```
FUNCTION Test()
        EnvPut("a", 123)
RESULT nothing

Test()
Dump(EnvGet("a"))
```

IsSet

IsSet(string)

Outputs whether the specified identifier is defined. This can be used for any identifier, including: variables, functions, and built-ins.

Type

Type(value)

Returns string representation of type for a value.

5.2 - CAST FUNCTIONS

ToInt

ToInt(value)

Converts a float or string number to an integer.

ToList

ToList(string)

Converts a string to a list of characters.

ToNum

ToNum(value)

Converts an integer or string number to a float.

ToStr

ToStr(number)

Converts a number to a string.

5.3 - LIST FUNCTIONS

Append

Append(list, value, ...)

Add values to the end of a list. Each value is Appended in the order supplied.

First

First(list)

Return the First value in a list. The list is unaffected. Returns (nothing) when list is empty.

Last

Last(list)

Return the Last value in a list. The list is unaffected. Returns (nothing) when list is empty.

Pop

Pop(list)

Remove and output the Last value in a list. Returns (nothing) when list is empty.

Prepend

Prepend(list, value, ...)

Add values to the beginning of a list. Each value is Prepended in the order supplied.

Range

Range(start, end)

Creates a list of numbers starting at start (s) and ending at end (e). The maximum Range length is 32767.

Remove

Remove(list, position)

Removes the value at position and resizes the list.

Shift

Shift(list)

Remove and return the First value in a list. Returns (nothing) when list is empty.

Size

Size(list)

Get the total Size of supplied list variable as a number.

5.4 - MAP FUNCTIONS

Index

Index(map, key, value)

Adds new key to map and stores value at that key. The key must not already exist.

GetKeys

GetKeys(map)

Returns the indices of a map as a list of strings.

Remove

Remove(map, key)

Removes the index identified by key and any value stored there.

5.5 - TEXT LAYER FUNCTIONS

Cls

Cls()

Clear the screen. Equivalent to pressing the CLEAR button on the interface.

Dump

Dump(value, ...)

Print any number of values and their types to the screen. Useful for debugging variables.

Write

Write(value, ...)

Prints any number of values to the screen without any additional spacing. Requires one or more arguments. Write inspects all possible variable types and can print string representations of lists, maps, and booleans.
When using Write() on long strings, any characters beyond the maximum screen width of 40 characters will not be displayed.

WriteLn

WriteLn(value, ...)

Similar to Write(), except that each output is written to its own line.

5.6 - GRID LAYER FUNCTIONS

ClearGrid

ClearGrid()

Instruct the system to clear the Grid screen layer at next opportunity.

DrawGrid

DrawGrid()

Instruct the system to apply any pending changes to the Grid screen layer at next opportunity.

Grid

Grid(x, y, r, g, b, a)

Set the Grid at coordinate (x, y) to Red (r), Green (g), Blue (b) color with Alpha (a) value.

5.7 - SPRITE LAYER FUNCTIONS

ClearSprites

ClearSprites()

Instruct the system to clear the Sprite screen layer at next opportunity.

DrawSprites

DrawSprites()

Instruct the system to apply any pending changes to the Sprite screen layer at next opportunity.

SpriteFlip

SpriteFlip(sprite, x, y)

Set the orientation of a sprite on the x and y axes. Valid axis values are -1 or 1.

Example:

```
# Left-facing sprite
SpriteFlip(0, -1, 1)

# Right-facing sprite
SpriteFlip(0, 1, 1)

# Right-side-up sprite
SpriteFlip(0, 1, 1)

# Upside-down sprite
SpriteFlip(0, 1, -1)
```

SpritePos

SpritePos(sprite, x, y)

Set the position of sprite to pixel coordinate (x, y). Valid coordinates are -64 to 320 on the x-axis and -64 to 200 on the y-axis. Negative positioning allows sprites to animate off the screen upwards (y) and to the left (x).

SpriteScale

SpriteScale(sprite, x, y)

Scale a sprite up or down using width(x) and height(y) scale multipliers. Valid scales are 0 - 7, where 0 is the original size of the sprite (8x8), and 7 is 8 times as large (64x64).

Scale	Ratio	Actual Size
0	1 to 1	8x8 pixels
1	2 to 1	16x16 pixels

2	3 to 1	24x24 pixels
3	4 to 1	32x32 pixels
4	5 to 1	40x40 pixels
5	6 to 1	48x48 pixels
6	7 to 1	56x56 pixels
7	8 to 1	64x64 pixels

SpriteSet

SpriteSet(sprite, x, y, r, g, b, a)

Set the sprite (s) at coordinate (x, y) to Red (r), Green (g), Blue (b) color with Alpha (a) value.

5.8 - PIXEL LAYER FUNCTIONS

ClearPixels

ClearPixels()

Instruct the system to clear the Pixel screen layer at the next opportunity.

DrawPixels

DrawPixels()

Instruct the system to apply any pending changes to the Pixel screen layer at next opportunity.

Pixel

Pixel(x, y, r, g, b, a)

Set the Pixel at coordinate (x, y) to Red (r), Green (g), Blue (b) color with Alpha (a) value.

5.9 - MEMORY FUNCTIONS

CacheGet

CacheGet(number)

Recall a previously stored function from cache memory.

CachePut

CachePut(number, function)

Store a defined function in cache memory. The function can be retrieved and invoked later. This is useful for sharing logic between programs.

MemGet

MemGet(register, position)

Returns the User Register value by string type and number index location.

Example:

```
# Get the number value from index 511
#  of the user number register.
MemGet("N", 511)

# Get the boolean value from index 64
# of the user boolean register.
MemGet("B", 64)

# Get the string value from index 149
# of user string register.
MemGet("S", 149)
```

MemPut

MemPut(register, position, value)

Sets the User Register value for type and index. User register values are not reset between program runs, and thus provide a mechanism for sharing data between programs. Note, that any values stored in User Register are disposed of during power off.

Example:

```
# Set the value 123 to index 511 of
# the user number register.
MemPut("N", 511, 123)

# Set the value TRUE to index 64 of
# the user boolean register.
MemPut("B", 64, true)

# Set the string "Hello World" to
# index 149 of user string register.
MemPut("S", 149, "Hello World")
```

SysGet

SysGet(register, position)

Returns the System Register value at position.

Example:

```
# Get the number value from index 511
# of the system number register.
SysGet("N", 511)

# Get the boolean value from index 64
# of the system boolean register.
SysGet("B", 64)

# Get the string value from index 149
# of system string register.
SysGet("S", 149)
```

SysPut

SysPut(register, position, value)

Sets the System Register value at position. System register values are initialized from ROM files during power on state. Any values modified in system ROMs are disposed of during power off.

Example:

```
# Set the value 123 to index 511 of
# the system number register.
SysPut("N", 511, 123)

# Set the value TRUE to index 64 of
# the system boolean register.
SysPut("B", 64, true)

# Set the string "Hello World" to
# index 149 of system string register.
SysPut("S", 149, "Hello World")
```

5.10 - AUDIO FUNCTIONS

Gate

Gate(channel, boolean)

Opens or closes the audio gate for a channel 0-3. When set to True, the gate is open. When set to False, the gate is closed. When the gate is open, the next available tone will begin to play. When the gate is closed, the playing note will enter the release stage, and no further notes will be triggered.

Tone

```
Tone(
    channel,
    shape,
    octave,
    note,
    attack,
    decay,
    sustain,
    release
)
```

Sets audio playback for a channel 0-3.

Volume

Volume(channel, level)

Sets the audio playback volume for a channel 0-3. Valid values are decibel amounts -80 to 0.

5.11 - FILE FUNCTIONS

DataRead

DataRead(file)

Read all lines from data file as a list.

Example:

```
SET list lines = DataRead("test")
EACH line IN lines REPEAT
        WriteLn(line)
DONE
```

DataReadLine

DataReadLine(file, number)

Read single line from data file at position. Position range is 0 to total number of lines - 1.

Example:

```
SET string line =
        DataReadLine("test", 0)
WriteLn(line)
```

DataWrite

DataWrite(file, mode, string)

Write a string to data file using specified mode.

Modes:

"A" - Append line to existing file.
"W" - Creates or overwrites existing file.

Example:

```
DataWrite("test", "W", "Hello")
DataWrite("test", "A", "World")
```

Data files are stored in user data directory with the .dat extension.

5.12 - INPUT FUNCTIONS

KeyDown

KeyDown()

Outputs the string for the keyboard key that is currently pressed. See Appendix 7: Input Keys for a list of supported keys.

MousePos

MousePos()

Outputs a map of the last recorded X and Y screen coordinates of the mouse position.

MouseButton

MouseButton(button)

Outputs boolean true when the specified mouse button is held, false otherwise. The following numbers are valid inputs for detecting buttons:

Button Number	Physical Button
0	Primary button (left click)
1	Secondary button (right click)

5.13 - STATUS WINDOW FUNCTIONS

Error

Error(string)

Display an error message and stop program execution.

Status

Status(title, body)

Set the Status window title and message while a program is running. System statuses and messages will take priority over anything added here, for example, errors, system state, and power/reset messages will display as usual.

5. 14 - MATH FUNCTIONS

Abs

Abs(number)

Return the absolute value for a given number.

Atan

Atan(number)

Return arctangent of a number.

Ceil

Ceil(number)

Return the nearest whole integer for a number, rounded up.

Cos

Cos(number)

Return cosine of number.

Floor

Floor(number)

Return the nearest whole integer for a number, rounded down.

Log

Log(number)

Return the natural logarithm for a number.

Pow

Pow(number, power)

Return the value of a number raised to the supplied power.

Random

Random(start, end)

Returns a Random number between start number and end number.

Round

Round(number)

Rounds a float number to the nearest whole integer number.

RoundPrec

RoundPrec(number, precision)

Rounds a float number to the specified decimal places. Valid precision values are 0-6 for that many decimal places. Specifying 0 is equivalent to Round(number), except that the internal float representation of the number is retained. Specifying a precision of 6 is equivalent to the maximum float precision supported by the system.

Example:

```
SET number pi = 3.14159265

Dump(
      pi,   # 3.141593
      RoundPrec(pi, 0), # 3
      RoundPrec(pi, 1), # 3.1
      RoundPrec(pi, 2), # 3.14
      RoundPrec(pi, 3), # 3.142
      RoundPrec(pi, 4), # 3.1416
      RoundPrec(pi, 5), # 3.14159
      RoundPrec(pi, 6)  # 3.141593
)
```

Sin

Sin(number)

Return sine of number.

Sqrt

Sqrt(number)

Return the square root of a number.

Tan

Tan(number)

Return tangent of number.

5.15 - MISC. UTILITY FUNCTIONS

Hash

Hash(value)

Encodes a number or string using a cryptographic hash function.

Seed

Seed(number)

Sets the seed used by the Random Number Generator. Setting this to a known value ensures the same set of Random numbers is returned every time.

Sync

Sync()

Call DrawGrid(), DrawPixels(), DrawSprites() simultaneously, then Wait(.1).

Example:

```
ClearGrid()
ClearSprites()
ClearPixels()
```

```
# Update all layers, wait 100ms.
Sync()
```

Timestamp

Timestamp()

Returns current UNIX timestamp, an integer containing the number of seconds since January 1, 1970 UTC.

Wait

Wait(seconds)

Pauses code evaluation for n amount of seconds.

Appendix 6: File Types

FREMScript

Code source files that can be run in the script editor.

Extension: *.frem*

Data Files

Data stored lines of text that can be read and written using the Data commands. See Chapter 14: Data Files.

Extension: *.dat*

Boot Program

Any *FREMScript* file can be loaded into the Boot Program slot using the Menu (F2). Boot Program files are automatically executed by the system during power on.

Extension: *.frem*

Text Files

Files for keeping notes, documentation.

Extension: *.txt*

Appendix 8: Introduction to Binary

Understanding binary numbers is key to understanding how computers and the electronics they are made of perform mathematical operations.

If you are new to binary, it is helpful to begin by taking a closer look at the number system we humans take for granted: decimal.

Breaking Down Decimal Numbers

The decimal counting system is based on powers of ten, hence the name: dec-imal. You'll also hear this system referred to as base10, which is a good way to refer to it, as you will see later.

In decimal (base10), we have the numbers 0 through 10 (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) to work with. Since humans typically have ten fingers, this is a convenient and non-coincidental number of digits to work with ("digit" derives from latin digitus meaning "finger or toe").

Let's take a closer look at a decimal number: 1234. By breaking 1234 into its powers of 10, we know that from right-to-left, we have a number in the thousands, hundreds, tens, and ones positions.

The following table illustrates this (note: the ^ exponentiation character means "to the power of"):

10^3	**10^2**	**10^1**	**10^0**
1	2	3	4

We can confirm the math with our understanding of decimal:

4 * 10^0 = 4 (10^0 = 1)
3 * 10^1 = 30 (10^1 = 10)
2 * 10^2 = 200 (10^2 = 100)
1 * 10^3 = 1000 (10^3 = 1000)

1000 + 200 + 30 + 4 = **1234**

That is: 4 ones + 3 tens + 2 hundreds + 1 thousands = 1234
As connoisseurs of decimal, we know we can continue on to higher powers: ten-thousands, hundred-thousands, millions, billions, trillions and onward to infinity.

You'll be pleased to learn that this concept of base10 applies to binary, or base2 also! The only difference, is rather than count in powers of 10, we count in powers of 2.

Breaking Down Binary Numbers

In binary, we have only two numbers to work with: 0 and 1. Because of this limitation, we don't break binary numbers down into powers of 10, but rather powers of 2. So, we can refer to binary as base2.

Let's break this binary number down in the same way we did earlier for decimal: 1101.

2^3	**2^2**	**2^1**	**2^0**
1	1	0	1

Breaking down the positions in powers of two:

1 * 2^0 = 1 (2^0 = 1)
0 * 2^1 = 0 (2^1 = 2)
1 * 2^2 = 4 (2^2 = 4)
1 * 2^3 = 8 (2^3 = 8)

8 + 4 + 0 + 1 = **13**

That is: 1 ones + 0 twos + 1 fours + 1 eights = 13

So, we've worked out that the binary number 1101 gives is the number 13 in decimal!

Remember, that in decimal, we derive the next higher powers as factors of 10, that is, to go from tens to hundreds, we multiply by 10. In binary, we get to the next power by a factor of two. So, the next after 8 is 16, then 32, 64, 128, and onward to infinity.

Let's break down some longer binary numbers, known as bytes. Bytes are simply a collection of 8 bits, which is a portmanteau of the words "binary digit". Therefore, a byte is just an 8-digit binary number.

Here's the binary representation of the decimal number 173: **1010 1101**

Note: we've broken it into two 4-bit chunks for readability.

Now, let's break the number down and mathematically derive its decimal equivalent:

2^7	**2^6**	**2^5**	**2^4**		**2^3**	**2^2**	**2^1**	**2^0**
1	0	1	0		1	1	0	1

1 * 2^0 = 1 (2^0 = 1)
0 * 2^1 = 0 (2^1 = 2)
1 * 2^2 = 4 (2^2 = 4)
1 * 2^3 = 8 (2^3 = 8)

0 * 2^4 = 0 (2^4 = 16)
1 * 2^5 = 32 (2^5 = 32)
0 * 2^6 = 0 (2^6 = 64)
1 * 2^7 = 128 (2^7 = 128)

128 + 0 + 32 + 0 + 8 + 4 + 1 = **173**

Let's try another one.
Here's a binary number: **1111 1111**
Again, we've broken it into two 4-bit chunks for readability.

Let's compute the decimal equivalent:

2^7	**2^6**	**2^5**	**2^4**		**2^3**	**2^2**	**2^1**	**2^0**
1	1	1	1		1	1	1	1

1 * 2^0 = 1 (2^0 = 1)
1 * 2^1 = 2 (2^1 = 2)
1 * 2^2 = 4 (2^2 = 4)
1 * 2^3 = 8 (2^3 = 8)

1 * 2^4 = 16 (2^4 = 16)
1 * 2^5 = 32 (2^5 = 32)
1 * 2^6 = 64 (2^6 = 64)
1 * 2^7 = 128 (2^7 = 128)

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255**

We've now computed the maximum value for an 8-bit number where all the bits are enabled ("1").

We can also say, “a byte holds a maximum of 255 distinct values.” As you work with *FREMScript*, you'll probably notice the range of 0-255 used in various built-in functions. This should indicate to you that the function is probably working with byte(s) of information behind the scenes.

Binary Math

We add base2 numbers in the same way we add base10 numbers.

Consider the following addition of two decimal numbers:

```
    25
+   17
=   42
```

Breaking down the math you likely did quickly in your head, your process probably went something like:

- Add 7 to 5, get 12.
- 12 exceeds 10, so keep the 2 and carry the 1 to the next column.
- Add 1 to 2 and then apply the carried 1 from the previous step, get 4.
- merge the columns, get answer: 42.

Think critically about what you are doing: you add until you reach the maximum for a particular column, which in base10 is "10". Then you carry the leftover to the next power (column) and continue the process. In base2, we do exactly the same, except that our maximum value is "1" in every column. Just like 9+1 in decimal is not "10", it's "0 carry 1", so goes that 1+1 in binary is "0 carry 1"

Let's add two 4-bit binary numbers:

```
    1011
+   0101
=   ????
```

Let's work through the math:

- Add the ones column, 1 + 1, get 0 carry 1.
- Add 0 and 1 in the twos column, apply the carry. Again, we have 1+1, so keep 0 and carry 1.
- Add 1 and 0 in the fours column, apply the previous carry and again get 1+1, keep 0 carry 1.
- In the final column we end up with 0+1 = 1 + the carry from the previous step. Now we have 0 carry 1.

- We drop the final carry as a new digit "1" in the sixteens column.
- Merge the columns, get the answer: 10000, or 16 in binary.

If we break down the binary numbers in the addition, we can see we have "1011" or 11 in decimal, and are adding "0101" or 5 in decimal. Since 11+5 = 16, our binary math checks out.

Subtraction with base2 is the same as base10 (this theme should be catching on by now: the math doesn't change just because the base number system changes).

Let's take a quick look at some binary subtraction using the same values from the previous Example:

```
    1011
-   0101
=   0110
```

Borrow from columns just like you would with decimal, keeping the following in mind: when you borrow in decimal, you get "10" to work with. That's 1 plus the maximum value (9) you can have in any decimal column. The same is true for binary subtraction, when you borrow, you're actually getting 1 plus the max of 1: 2-1 when borrowing, is 1.

The result of our subtraction is 0110 in base2, which is 5 in base10.

This should give you enough information to work comfortably with binary values going forward, but there is much more to learn on the topic.

Here are some topics you can research on your own at your local library:

- One and Two's complement (how negative numbers and subtraction work in computers)
- Binary Multiplication and Division
- Bitwise operations (AND, OR, XOR, NOT, SHIFT)
- Bit masking
- Other base number systems common in computing: Octal (base8), Hexadecimal (base16)

Why do Computers use Binary?

You've learned quite a bit (pun intended) about binary, but you are almost certainly wondering, why do computers count this way? It's a fair question. For humans, it would be decidedly more convenient if computers just counted in powers of ten, wouldn't it?

The basic answer is in how the computer hardware works. The internal components: integrated circuits, transistors, diodes, etc. all contribute to the complex operations that make the computer work. The electrical nature and tolerances in manufacturing of these components means that the reliable states limited to "on" and "off" are fundamental to their operation. Since "On" and "Off" can be represented with "1" and "0", respectively, binary is a logical choice to represent them.

The computer, being made up of various combinations of such components, naturally inherits the binary system. The instructions that you, the programmer, provide to the computer ultimately boil down to interactions with these fundamental components at the electrical level.