GEMscript

Language Reference
2015

Table of Contents

GEMscript

The programming language of GEMstudio is called GEMscript. It is used to write utility functions and event handlers to service user and system events. The syntax of GEMscript was influenced by JavaScript. As a result, GEMscript is a multi-paradigm language that supports object-oriented, imperative, and functional programming styles.

Applications built in GEMstudio have two components: (1) the graphical user interface, and (2) the program logic. GEMstudio provides a visual GUI builder that lets you build your application's user interface without any (or very little) programming. At runtime, each user interface object can generate events in response to the user interacting with the object. GEMscript is used to write the software that responds to these user events. So, the program logic in your application is defined by the event handlers and utility functions that you write in the GEMscript programming language.

GEMscript vs JavaScript vs C

GEMscript syntax is similar to JavaScript (whose syntax was influenced by C) but they are not identical. Here you will find some key differences.

• There are no structures or unions in GEMscript. For a "struct" substitute, see symbolic subscripts for arrays.

• The accepted syntax for rational numbers is stricter than that of floating point values in C. Values like ".5" and "6." are acceptable in C, but in GEMscript, "0.5" and "6.0" must be used, respectively. In C, the decimal period is optional if an exponent is included, so "2E8" can be used; GEMscript does not accept the upper case "E" (use a lower case "e") and it requires the decimal point: e.g. "2.0e8". See Constants - Rational Numbers for more.

• GEMscript does not provide "pointers". For the purpose of passing function arguments by reference, GEMscript provides a "reference" argument. See Calling Functions - Function arguments. The "placeholder" argument replaces some uses of the NULL pointer. See Default values of function arguments.

• Numbers can have hexadecimal, decimal or binary radix. Octal radix is not supported. See General Syntax - Constants. Hexadecimal numbers must start with "0x" (a lower case "x"), the prefix "0X" is invalid.

• Escape sequences ("\n", "\t", etc.) are the same, except for "\ddd" where "ddd" represent three decimal digits, instead of the octal digits that C/C++ uses. The backslash ("\") may be replaced with another symbol.

• Cases in a `switch` statement are not "fall through". Only a single statement may (and must) follow each `case` label. To execute multiple instructions, compound statements must be used, meaning braces must be wrapped around the code. The `default` clause of a `switch` statement must be the last clause of the `switch` statement. Cases in GEMscript are structured "if" statements.

• A `break` statement breaks out of loops only. In C/C++, the break statement also ends a `case` in a `switch` statement. Switch statements are implemented differently in GEMscript (see the switch statement)

• GEMscript supports "array assignment", with the restriction that both arrays must have the same size. For example, if "a" and "b" are both arrays with 6 cells, the expression "a = b" is valid. Next to literal strings, GEMscript also supports literal arrays, allowing the expression "a = {0,1,2,3,4,5}" (where "a" is an array variable with 6 elements).

• `defined` is an operator, not a preprocessor directive. The `defined` operator in GEMscript operates on constants (declared with `const`), global variables, local variables and functions.

• The `sizeof` operator returns the size of a variable in "elements", not in "bytes". In a multidimensional array, this element is the number of sub-arrays in a particular dimension. In the case of single dimension arrays, including basic strings, this is the length of the array in 32-bit elements. See `sizeof` in Miscellaneous for details.

• The empty instruction is an empty compound block, not a semicolon (See Empty statement). This modification avoids a frequent error.

• The compiler directives differ from C's preprocessor commands. Notably, the #define directive is incompatible with that of C/C++, and #ifdef and #ifndef are replaced by the more general #if directive (see Directives). To create numeric constants, see Symbolic Constants; to create string constants, see String Constants.

• Text substitutions (preprocessor macros; see the #define directive) are not matched across lines. That is, the text that you want to match and replace with a #define macro must appear on a single line.

• A division is carried out in such a way that the remainder after division has (or would have) the same sign as the denominator. For positive denominators, this means that the direction for truncation for the operator "/" is always towards the smaller value, where -2 is smaller than -1, and that the "%" operator always gives a positive result —regardless of the sign of the numerator. See Arithmetic.

• There is no unary "+" operator, which is a "no-operation" operator anyway.

• Three of the bitwise operators have different precedence than in C. The precedence levels of the "&", "^" and | operators is higher than the relational operators.

• The "extern" keyword does not exist in GEMscript; the current implementation of the compiler has no "linking phase". The GEMscript compiler can optimize out functions and global variables that you do not use.

• The keyword `const` in GEMscript implements the enum functionality from C, see Symbolic Constants.

• In most situations, forward declarations of functions (i.e., prototypes) are not necessary. GEMscript is a two-pass compiler, it will see all functions on the first pass and use them in the second pass. User-defined operators must be declared before use, however. If provided, forward declarations must match exactly with the function definition, parameter names may not be omitted from the prototype or differ from the function definition. GEMscript cares about parameter names in prototypes because of the "named parameters" feature. One uses prototypes to call forwardly declared functions. When doing so with named parameters, the compiler must already know the names of the parameters (and their position in the parameter list). As a result, the parameter names in a prototype must be equal to the ones in the definition.

Storing Values in Variables

GEMscript has a few different types of variables: Integer, Floating Point, and Strings, as well as single- and multi-dimensional arrays.

The keyword new declares a new variable. This is followed by the type: int, float, or string. If there is no type, int is assumed. Then comes the name of the variable, which must follow the rules of any Identifier:

Identifiers consist of the characters a. . . z, A. . . Z, 0. . . 9, _ or @; the first character may not be a digit. The characters @ and _ by themselves are not valid identifiers.

For special declarations, the keyword new is replaced by static, or public.

Unless it is explicitly initialized, the value of the new variable is zero. A variable declaration may occur:
• at any position where a statement would be valid —local variables;
• at any position where a function declaration or a function implementation would be valid —global variables;
• in the first expression of a for loop instruction —also local variables.

The range of an int is a signed 32-bit value: -2147483648 to 2147483647.
The float type is a single precision 32-bit format which has a mantissa of about 7.22 decimal and a maximum exponent of about 38.23 decimal.
Strings are stored in the UTF-8 encoding format.

Initialization

Data objects can be initialized at their declaration. The initializer of a global data object must be a constant. Arrays, global or local, must also be initialized
with constants. Uninitialized data defaults to zero.

Examples:

```
new int i = 1;
new int j;                      // j is zero
new j2;                         // defaults to int type
new int k = 'a';                // k has character code for letter
'a'
new int a[] = [1,4,9,16,25];    // a has 5 elements
new int s1[20] = ['a','b'];     // the other 18 elements are 0
new string s2{} = "Hello world..."; // a string
```

Examples of invalid declarations:

```
new int c[3] = 4;               // an array cannot be set to a
value
new int i = "Good-bye";         // only an array can hold a string
new int q[];                    // unknown size of array
new int p[2] = { i + j, k - 3 }; // array initializers must be
constants
```

Local Declarations

A local declaration appears inside a compound statement. A local variable can only be accessed from within the compound statement and from nested compound statements. A declaration in the first expression of a for loop instruction is also a local declaration.

Global Declarations

A global declaration appears outside a function and a global variable is accessible to any function inside <script> tags on that current page or any child page. Global data objects can only be initialized with constant expressions.

The term global is a misnomer here, since the only memory that persists from page to page is InternalRAM, but there are some implications when dealing with nested pages.

Static local declarations

A local variable is destroyed when the execution leaves the compound block in which the variable was created. Local variables in a function only exist during the run time of that function. Each new run of the function creates and initializes new local variables. When a local variable is declared with the keyword static rather than new, the variable remains in existence after the end of a function. This means that static local variables provide private, permanent storage that is accessible only from a single function (or compound block). Like global variables, static local variables can only be initialized with constant expressions.

Static global declarations

In GEMstudio there is no difference between a new global variable and a static global variable.

Public declarations

Global "simple" variables (no arrays) may be declared public in two ways:

• declare the variable using the keyword public instead of new
• start the variable name with the "@" symbol. (when referencing the variable, use the @ as well)

Example:
```
public int last_error;
```

Public variables behave like global variables, with the addition that objects outside of the script can also read and write public variables. A (normal) global variable can only be accessed by the functions in your script — the rest of the Amulet OS is unaware of them.

A possible use is that the Amulet could ask that you declare a variable with a specific name as public for special purposes —such as the most recent error number, or the general program state. These special purposes will be described in further detail when they become available.

Constant variables

It is sometimes convenient to be able to create a variable that is initialized once and that may not be modified. Such a variable behaves much like a symbolic constant, but it still is a variable. To declare a constant variable, insert the keyword `const` between the keyword that starts the variable declaration —new, static or public— and the variable type. Examples:

```
new const int address[4] = { 192, 168, 0, 66 };
public const int status; /* initialized to zero */
```

Three typical situations where a constant variable may be used are:
• To create a string or array constant; symbolic constants cannot be indexed.
• A public variable that should be set by the Amulet OS, and only by the Amulet OS. See the preceding section for public variables.
• A special case is to mark array arguments to functions as const. Array arguments are always passed by reference, declaring them as const guards against unintentional modification.

Arrays

GEMscript supports single and multidimensional arrays. It is possible to have arrays of any type, such as integer or floating point. Strings are also implemented as arrays, but are packed into the smallest space required for UTF-8 encoding and are addressable on the byte level, while all other types of array elements are only addressable as 32-bit data.

Single Dimension

The syntax `name[constant]` declares name to be an array of "constant" elements, where each element is a single cell of integers or floats. The name is a placeholder of an identifier name of your choosing and constant is a positive non-zero value; constant may be absent. If there is no value between the brackets, the number of elements is set equal to the number of initializers —see the example below. The array index range is "zero based" which means that the first element is at name[0] and the last element is name[constant-1].

The syntax `name{constant}` also declares name as an array of constant elements, but these elements can be indexed on the byte level. This is usually to support string manipulation. The array is always a multiple of 4 bytes, so `constant` will be rounded up to the nearest multiple of 4.

Progressive initializers

The ellipsis operator continues the progression of the initialization constants for an array, based on the last two initialized elements. The ellipsis operator (three dots, or "...") initializes the array up to its declared size.

Examples:

```
new int a[10] = { 1, ... };          // sets all ten elements to 1
new int b[10] = { 1, 2, ... };       // b = 1, 2, 3, 4, 5, 6, 7,
8, 9, 10
new int c[8]  = { 1, 2, 40, 50, ... }; // c = 1, 2, 40, 50, 60, 70,
80, 90
new int d[10] = { 10, 9, ... };      // d = 10, 9, 8, 7, 6, 5, 4,
3, 2, 1
```

Symbolic subscripts

An array may be declared with a list of symbols instead of a value for its size. An individual subscript may also be interpreted as a sub-arrays. The sub-array syntax applies as well to the initialization of an array with symbolic subscripts. To initialize a "message" array with fixed values for a messagae type and message priority, the syntax is:

```
new int msg[.type, .priority] = { 0, 1 };
```

The default type of the subscripts is defined as an int. The initializer consists of two integer values; these go into the fields ".text" and ".priority" respectively. An array dimension that is declared as a list of symbolic subscripts may only be indexed with these subscripts. From the above declaration of variable "msg", we may use:

```
new int type = msg[.type];
msg[.priority] = 10 - msg[.priority];
```

It is an error, however, to use a (numeric) expression to index "msg". For example, "msg[1]" is an invalid expression. Since an array with symbolic subscripts may not be indexed with an expression, the square brackets that enclose the expression become optional. These brackets may be omitted. The snippet below is equivalent to the previous snippet.

```
new int type = msg.type;
msg.priority = 10 - msg.priority;
```

A subscript may have an explicit type as well. This type will then override the default type for array elements. In the declaration in the snippet below, the expression "field.type" is a plain integer, but the expression "field.value" has floating point type, and the expression "field.word" is a string.

```
new int field[ .type, float .value, string .word{20} ];
```

Multi-dimensional arrays

Multi-dimensional arrays are arrays that contain references to the sub-arrays. That is, a two-dimensional array is an "array of single-dimensional arrays". Below are a few examples of declarations of two-dimensional arrays.

```
new int a[4][3];
new int b[3][2] = [ [ 1, 2 ], [ 3, 4 ], [ 5, 6 ] ];
new int c[3][3] = [ [ 1 ], [ 2, ...], [ 3, 4, ... ] ];
new string d[2]{10} = [ "agreement", "dispute" ];
new string e[2]{} = [ "OK", "Cancel" ];
new string f[]{} = [ "OK", "Cancel" ];
```

As the last two declarations (variables "e" and "f") show, the final dimension of an array may have an unspecified length, in which case the length of each sub-array is determined from the related initializer. Every sub-array may have a different size; in this particular example, "e[1][5]" contains the letter "l" from the word "Cancel", but "e[0][5]" is invalid because the length of the sub-array "e[0]" is only three cells (containing the letters "O", "K" and a zero terminator).

The difference between the declarations for arrays "e" and "f" is that we let the compiler count the number of initializers for the major dimension —"sizeof f" is 2, like "sizeof e" (see the next section on the sizeof operator).

The sizeof operator

The `sizeof` operator returns the size of a variable in 32-bit "elements" called cells. For a simple (non-compound) variable, the result of `sizeof` is always 1, because an element is either a 32-bit integer or 32-bit floating point value for a simple variable. An array with one dimension holds a number of 32-bit cells and the `sizeof` operator returns that number. In the snippet below, the variable `length` would therefore contain 4 because the array "msg" holds 4 integers with.

```
new int msg[] = {1, 2, 3, 4};
new int length = sizeof(msg);  // length = 4
```

The sizeof operator always returns the number of cells, even for a string. That is, in the next snippet, the value assigned to `length` would be less than "4" —although there are five bytes in the array (four 1-byte characters plus a terminating null byte), those are packed in fewer cells.

```
new string msg{} = "Help";
new int length = sizeof(msg);  // length = 2
```

With multi-dimensional arrays, the `sizeof` operator can return the number of elements in each dimension. For the last (minor) dimension, an element will again be a cell, but for the major dimension(s), an element is a sub-array. In the following code snippet, observe that the syntax `sizeof matrix` refers to the major dimension of the two-dimensional array and the syntax `sizeof matrix[]` refers to the minor dimension of the array. The values that this snippet prints are 3 and 2 (for the major and minor dimensions respectively):

```
new int matrix[3][2] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
new int length1 = sizeof(matrix);   // length1 = 3
new int length2 = sizeof(matrix[]); // length2 = 2
```

The application of the `sizeof` operator on multi-dimensional arrays is especially convenient when used as a default value for function arguments.

Operators and expressions

An operator is a program element that is applied to one or more operands in an expression or statement. Operators that take one operand, such as the increment operator (++), are referred to as unary operators. Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators. One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in GEMscript.

Notational conventions

The operation of some operators depends on the specific kinds of operands. Therefore, operands are notated thus:

e   any expression;

v   any expression to which a value can be assigned ("lvalue" expressions);

a   an array;

f   a function;

s   a symbol —which is a variable, a constant or a function.

Expressions

An expression consists of one or more operands with an operator. The operand can be a variable, a constant or another expression. An expression followed by a semicolon is a statement. Examples of expressions:

```
v++
f(a1, a2)
v = (ia1 * ia2) / ia3
```

Arithmetic

Arithmetic operators can be used with both Integer and Floating Point expressions. Mixed types in binary operators will result in a conversion of the `int` argument to a `float` prior to the operation.

Operator	Usage/Result
+	`e1 + e2`
	Results in the addition of `e1` and `e2`.
−	`e1 - e2`
	Results in the subtraction of `e1` and `e2`.
	`-e`
	Results in the arithmetic negation of `e` (two's complement).
*	`e1 * e2`
	Results in the multiplication of `e1` and `e2`.
/	`e1 / e2`
	Results in the division of `e1` by `e2`. The result is truncated to the nearest integral value that is less than or equal to the quotient. Both negative and positive values are rounded down, i.e. towards −∞.
%	`e1 % e2`
	Results in the remainder of the division of `e1` by `e2`. The sign of the remainder follows the sign of `e2`. Integer division and remainder have the Euclidean property: D = q*d + r, where q = D/d and r = D%d.
++	`v++`
	increments `v` by 1; the result if the expression is the value of `v` before it is incremented.
	`++v`
	increments `v` by 1; the result if the expression is the value of `v` after it is incremented.
−−	`v--`
	decrements `v` by 1; the result if the expression is the value of `v` before it is decremented.
	`--v`
	decrements `v` by 1; the result if the expression is the value of `v` after it is decremented.

Notes: The unary + is not defined in GEMscript. The operators ++ and −− modify the operand. The operand must be an lvalue.

Bit manipulation

Bitwise operators are not available to Floating Point expressions.

~	`~e`
	results in the one's complement of `e`.
>>	`e1 >> e2`
	results in the arithmetic shift to the right of `e1` by `e2` bits. The shift operation is signed: the leftmost bit of `e1` is copied to vacant bits in the result.
>>>	`e1 >>> e2`
	results in the logical shift to the right of `e1` by `e2` bits. The shift operation is unsigned: the vacant bits of the result are filled with zeros.
<<	`e1 << e2`
	results in the value of `e1` shifted to the left by `e2` bits; the rightmost bits are set to zero. There is no distinction between an arithmetic and a logical left shift
&	`e1 & e2`
	results in the bitwise logical "and" of `e1` and `e2`.
|	`e1 | e2`
	results in the bitwise logical "or" of `e1` and `e2`.
^	`e1 ^ e2`
	results in the bitwise "exclusive or" of `e1` and `e2`.

Assignment

The result of an assignment expression is the value of the left operand after the assignment.

=	`v = e`
	assigns the value of `e` to variable `v`.
	`v = a`
	assigns array `a` to variable `v`; `v` must be an array with the same size and dimensions as `a`; `a` may be a string or a literal array.

Note: the following operators combine an assignment with an arithmetic or a bitwise operation; the result of the expression is the value of the left operand after the arithmetic or bitwise operation. The compiler will expand this to two separate operations. Bitwise assignments are not valid for Floating Point expressions.

`+=`	`v += e`
	increments `v` with `e`.
`-=`	`v -= e`
	decrements `v` with `e`
`*=`	`v *= e`
	multiplies `v` with `e`
`/=`	`v /= e`
	divides `v` by `e`.
`%=`	`v %= e`
	assigns the remainder of the division of `v` by `e` to `v`.
`>>=`	`v >>= e`
	shifts `v` arithmetically to the right by `e` bits.
`>>>=`	`v >>>= e`
	shifts `v` logically to the right by `e` bits.
`<<=`	`v <<= e`
	shifts `v` to the left by `e` bits.
`&=`	`v &= e`
	applies a bitwise "and" to `v` and `e` and assigns the result to `v`.
`|=`	`v |= e`
	applies a bitwise "or" to `v` and `e` and assigns the result to `v`.
`^=`	`v ^= e`
	applies a bitwise "exclusive or" to `v` and `e` and assigns the result to `v`.

Relational

Relational operators can be used with both Integer and Floating Point expressions. Mixed types in binary operators will result in a conversion of the `int` argument to a `float` prior to the operation.

A logical "false" is represented by an integer value of 0; a logical "true" is represented by any value other than 0. Value results of relational expressions are either 0 or 1.

==	`e1 == e2`
	results in a logical "true" if `e1` is equal to `e2`.
!=	`e1 != e2`
	results in a logical "true" if `e1` differs from `e2`.

Note: the following operators may be "chained", as in the expression "e1 <= e2 <= e3", with the semantics that the result is "1" if all individual comparisons hold and "0" otherwise.

<	`e1 < e2`
	results in a logical "true" if `e1` is smaller than `e2`.
<=	`e1 <= e2`
	results in a logical "true" if `e1` is smaller than or equal to `e2`.
>	`e1 > e2`
	results in a logical "true" if `e1` is greater than `e2`.
>=	`e1 >= e2`
	results in a logical "true" if `e1` is greater than or equal to `e2`.

Boolean

These arithmetic operators can be used with both Integer and Floating Point expressions. Mixed types in binary operators will result in a conversion of the `int` argument to a `float` prior to the operation.

A logical "false" is represented by an integer value of 0; a logical "true" is represented by any value other than 0. Value results of Boolean expressions are Integers, either 0 or 1.

`!`	`!e`
	results to a logical "true" if `e` was logically "false".
`||`	`e1 || e2`
	results to a logical "true" if either `e1` or `e2` (or both) are logically "true". The expression `e2` is only evaluated if `e1` is logically "false".
`&&`	`e1 && e2`
	results to a logical "true" if both `e1` and `e2` are logically "true". The expression `e2` is only evaluated if `e1` is logically "true".

Note: Floating point uses a signed zero. Negative zero reports a logical false. If you "logically OR" a positive zero with a negative zero, the result will be false, yet if you "logically AND" positive zero and negative zero, the result is also false.

Miscellaneous

[ ]	`a[e]`
	array index: results to cell `e` from array `a`.
{ }	`a{e}`
	string index: results to byte `e` from "packed" array `a`.
( )	`f(e1,e2,...eN)`
	results to the value returned by the function `f`. The function is called with the arguments `e1, e2, . . . eN`. The order of evaluation of the arguments is undefined (an implementation may choose to evaluate function arguments in reversed order).
? :	`e1 ? e2 : e3`
	results in either `e2` or `e3`, depending on the value of `e1`. The conditional expression is a compound expression with a two part operator, "`?`" and "`:`". Expression `e2` is evaluated if `e1` is logically "true", `e3` is evaluated if `e1` is logically "false".
,	`e1, e2`
	results in `e2`, `e1` is evaluated before `e2`. If used in function argument lists or a conditional expression, the comma expression must be surrounded by parentheses.
defined	`defined s`
	results in the value 1 if the symbol is defined. The symbol may be a constant or a global or local variable.
sizeof	`sizeof s`
	results in the size in "elements" of the specified variable. For simple variables and for arrays with a single dimension, an element is a 32-bit cell. For multi-dimensional arrays, the result is the number of array elements in that dimension —append `[]` to the array name to indicate a lower/more minor dimension. If the size of a variable is unknown, the result is zero. When used in a default value for a function argument, the expression is evaluation at the point of the function call, instead of in the function definition.

Operator precedence

The table beneath groups operators with equal precedence, starting with the operator group with the highest precedence at the top of the table. If the expression evaluation order is not explicitly established by parentheses, it is determined by the association rules. For example: a*b/c is equivalent with (a*b)/c because of the left-to-right association, and a=b=c is equivalent with a=(b=c).

`()`	function call	left-to-right
`[]`	array index (cell)	
`{}`	array index (character)	
`!`	logical not	right-to-left
`~`	one's complement	
`–`	two's complement (unary minus)	
`++`	increment	
`--`	decrement	
`:`	tag override	
`defined`	symbol definition status	
`sizeof`	symbol size in "elements"	
`*`	multiplication	left-to-right
`/`	division	
`%`	remainder	
`+`	addition	left-to-right
`–`	subtraction	
`>>`	arithmetic shift right	left-to-right
`>>>`	logical shift right	
`<<`	shift left	
`&`	bitwise and	left-to-right
`^`	bitwise exclusive or	left-to-right
`|`	bitwise or	left-to-right
`<`	smaller than	left-to-right
`<=`	smaller than or equal to	
`>`	greater than	
`>=`	greater than or equal to	
`==`	equality	left-to-right
`!=`	inequality	
`&&`	logical and	left-to-right
`||`	logical or	left-to-right
`?:`	conditional	right-to-left
`= *= /= %= += -= >>= >>>= <<= &= ^= |=`	assignment	right-to-left
`,`	comma	left-to-right

Code flow control statements

A statement may take one or more lines, whereas one line may contain two or more statements. Control flow statements (if, if–else, for, while, do–while and switch) may be nested.

Statement label

Usage:
label:

A label consists of an identifier followed by a colon (":"). A label is a "jump target" of the goto statement. Each statement may be preceded by a label. There must be a statement after the label; an empty statement is allowed. The scope of a label is the function in which it is declared (a goto statement cannot therefore jump out off the current function to another function).

Example:

```
nestedLoops(int maxA, int maxB)
{
    new int a, b, c;
    for (a=0; a <= maxA; a++)
    {
        for (b=0; b <= maxB; b++)
        {
            if ((a * b) > 256)
            {
                goto end;      // if larger than a byte, get out of
both loops
            }
            c = a* b;
        }
    }
    end:                       // goto will jump to this label
    internalRAM.byte(0x0a) = a;
    internalRAM.byte(0x0b) = b;
    internalRAM.byte(0x0c) = c;
}
```

Compound statement

Usage:
```
{
    statement1;
    statement2;
    .
    .
    statementN;
}
```

A compound statement is a series of zero or more statements surrounded by braces ({ and }). The final brace (}) should not be followed by a semicolon. Any statement may be replaced by a compound statement. A compound statement is also called a block. A compound statement with zero statements is a special case, and it is called an "empty statement".

Example:

```
if ((a + b) > c)
{
    c -= (a + b);   // compound statement line #1
    b++;            // compound statement line #2
    a += 2;         // compound statement line #3
}
```

Expression statement

Usage:
statement;

Any expression becomes a statement when a semicolon (";") is appended to it. An expression also becomes a statement when only white space follows it on the line and the expression cannot be extended over the next line.

Example:

```
a = b + c;      // simple expression statement
```

Empty statement

Usage:
{}

An empty statement performs no operation and consists of a compound block with zero statements; that is, it consists of the tokens "{ }". Empty statements are used in control flow statements if there is no action or when defining a label just before the closing brace of a compound statement. An empty statement does not end with a semicolon.

Example:

```
//do nothing while isReady() returns false
while (!isReady()) {}
```

assert

Usage:
assert (*expression);*

Aborts script execution if the expression evaluates to logically "false". Keep in mind that **assert** aborts the entire script, not just the local function.

Example:

```
fibonacci(int n)
{
    assert(n > 0);              // if n is 0 or negative, abort the
script
    new int a = 0, b = 1;

    for (new int i = 2; i < n; i++)
    {
        new int c = a + b;
        a = b;
        b = c;
    }
    return a + b;
}
```

break

Usage:
break;

Terminates and exits the smallest enclosing do, for or while statement from any point within the loop other than the logical end. The break statement moves program control to the next statement outside the loop.

Example:

```
for (new int i=0; i<10; i++)
{
    newCh = s{i++};
    if (newCh < 0x20)
        break;                  // if a non-ASCII character encountered,
break out of the loop
    newString{j++} = newCh;// build new string with lowercase ASCII
letters
}
```

continue

Usage:
continue;

Terminates the current iteration of the smallest enclosing do, for, or while statement and moves program control to the condition part of the loop. If the looping statement is a for statement, control moves to the third expression in the for statement (and thereafter to the second expression). Most functions can be written without the need for a **continue** statement, but it can be useful in the right situation.

Example:

```
do
{
    newCh = s{i++};
    if (newCh < 0x61 || newCh > 0x7B)
        continue;   // only include lower case ASCII letters
    newString{j++} = newCh;// build new string with lowercase ASCII
letters
} while (newCh);
```

do-while

Usage:
do
{
   statement; (or compound statement)
} **while** (*expression*);

Executes a statement before the condition part (the while clause) is evaluated. The statement is repeated while the condition is logically "true". The statement is executed at least once.

Example:

```
int factorial(int n)
{
    new int f = 1;

    do
    {
        f *= n--;
    } while (n > 0);
    return f;
}
```

exit

Usage:
exit;

Abort the script, immediately returning control to the Amulet OS. Keep in mind that **exit** aborts the entire script, not just the local function.

Example:

```
#define NO_ERROR = 0;
new int error = NO_ERROR;

...

if (error != NO_ERROR)
     exit;
```

for

Usage:
for (*expression1* ; *expression2* ; *expression3*)
{
   statement; (or compound statement)
}

All three expressions are optional.

expression1: Evaluated only once, and before entering the loop. This expression may be used to initialize a variable. This expression may also hold a variable declaration, using the new syntax. A variable declared in this expression exists only in the for loop. You cannot combine an expression (using already existing variables) and a declaration of new variables in this field — either all variables in this field must already exist, or they must all be created in this field.

expression2: Evaluated before each iteration of the loop and ends the loop if the expression results to logically "false". If omitted, the result of expression 2 is assumed to be logically "true".

expression3: Evaluated after each execution of the statement. Program control moves from expression 3 to expression 2 for the next (conditional) iteration of the loop. The statement for(; ;) is equivalent with while (true).

Example:

```
int factorial(int n)
{
    new int f = 1;

    for (new int i = 2; i <= n; i++)
    {
        f = f*i;
    }
    return f;
}
```

goto

Usage:
goto *label*

Moves program control (unconditionally) to the statement that follows the specified label. The label must be within the same function as the goto statement (a goto statement cannot jump out of a function).

Example:

```
nestedLoops(int maxA, int maxB)
{
    new int a, b, c;
    for (a=0; a <= maxA; a++)
    {
        for (b=0; b <= maxB; b++)
        {
            if ((a * b) > 256)
            {
                goto end;      // if larger than a byte, get out of
both loops
            }
            c = a* b;
        }
    }
    end:                       // goto will jump to this label
    internalRAM.byte(0x0a) = a;
    internalRAM.byte(0x0b) = b;
    internalRAM.byte(0x0c) = c;
}
```

if

Usage:
if (*expression*)
{
   statement 1;
}
else
{
   statement 2;
}

Executes statement 1 if the expression results to logically "true". The else clause of the if statement is optional. If the expression results to logically "false" and an else clause exists, the statement associated with the else clause (statement 2) executes. When if statements are nested and else clauses are present, a given else is associated with the closest preceding if statement in the same block.

Example:

```
// returns: 2 if ch is ASCII letter,
//          1 if ch is ASCII number,
//          0 if ch is anything else
int AlphaOrNumeric(int ch)
{
    if (('A' <= ch <= 'Z') || ('a' <= ch <= 'z'))
    {
        return 2;  // ASCII letter
    }
    else if ('0' <= ch <= '9')
    {
        return 1;  // ASCII number
    }
    else
    {
        return 0;  // not ASCII number or letter
    }
}
```

return

Usage:
return *expression;*

Terminates the current function and moves program control to the statement following the calling statement. The value of the expression is returned as the function result. The expression may be an array variable or a literal array. The expression is optional, but it must start on the same line as the return statement if it is present. If absent, the value of the function is zero.

Example:

```
string buildString(int ch)
{
    new String newStr{100};
    new String tmpCh{4};

    if (('A' <= ch <= 'Z') || ('a' <= ch <= 'z'))
    {
        tmpCh{0} = ch;
        newStr = append("The character is a valid ASCII character
(",tmpCh,")");
    }
    else
    {
        newStr = append("The character is NOT a valid ASCII
character (",itoa(ch),")");
    }

    return newStr;
}
```

sleep

Usage:
sleep *expression*;

This is the same as the [exit statement](#).

switch

Usage:
switch (*expression*)
{
  case list
  .
  .
  .
}

Transfers control to different statements within the switch body depending on the value of the switch expression. The body of the switch statement is a compound statement, which contains a series of "case clauses". Each "case clause" starts with the keyword case followed by a constant list and one (and only one) statement.

The constant list is a series of expressions, separated by commas, that each evaluates to a constant value. The constant list ends with a colon. To specify a "range" in the constant list, separate the lower and upper bounds of the range with a double period (".."). An example of a range is: "case 1..9:".

The switch statement moves control to a "case clause" if the value of one of the expressions in the constant list is equal to the switch expression result. The "default clause" consists of the keyword default and a colon. The default clause is optional, but if it is included, it must be the last clause in the switch body. The switch statement moves control to the "default clause" if none of the case clauses match the expression result.

Note: Unlike C and other languages, GEMscript switch statements do NOT use the break command to end a case. One, and only one, statement is allowed per case. If you need multiple lines of code you must either call a function of your own creation that contains the multiple lines of code, or wrap your lines in braces to form a compound statement.

Example:

```
public setWeekday()
{
    // this assumes InternalRAM.byte(0) currently holds the day of
the week
    switch (InternalRAM.byte(0))
    {
        case 0, 1:                  /* 0 = Saturday, 1 = Sunday */
            print("weekend");
        case 2:
            print("Monday");
        case 3:
            print("Tuesday");
        case 4:
            print("Wednesday");
        case 5:
            print("Thursday");
        case 6:
```

```
            print("Friday");
        case 7..9:
            print("error: single digit > 6");
        case 10..99:
            print("error: double digit");
        default:
            print("error: triple digit");
    }
}

public print(str{})
{
    InternalRAM.string(0) = str;
    Amulet:document.MyString_1.forceUpdate();
}
```

while

Usage:
while (*expression*)
{
   statement; (or compound statement)
}

Evaluates the expression and executes the statement if the expression result yields logically "true". After the statement has executed, program control returns to the expression again. The statement is thus executed while the expression is true.

Example:

```
// iterate through until greatest common denominator is found
while (b != 0)
{
    if (a > b)
        a = a - b;
    else
        b = b - a;
}
```

General Syntax

The syntax of GEMscript is very C-like with JavaScript Document Object Model (DOM) influences. You will see the DOM syntax mostly in the calls to Amulet OS data structures (widgets, InternalRAM)

Format

Identifiers, numbers and tokens are separated by spaces, tabs, carriage returns and "form feeds".
Series of one or more of these separators are called white space.

Optional Semicolons

Semicolons (to end a statement) are optional if they occur at the end of a line. Semicolons are required to separate multiple statements on a single line. An expression may still wrap over multiple lines, but postfix operators (++ and --) must appear on the same line as their operand. All of the examples in this document use semicolons to end each statement.

Comments

Text between the tokens /* and */ (both tokens may be at the same line or at different lines) and text behind // (up to the end of the line) is a programming comment. The parser treats a comment as white space. Comments may not be nested.

Identifiers

Names of variables, functions and constants. Identifiers consist of the characters a. . . z, A. . . Z, 0. . . 9, _ or @; the first character may not be a digit. The characters @ and _ by themselves are not valid identifiers, i.e. "_Up" is a valid identifier, but "_" is not. GEMscript is case sensitive.

The parser will truncate an identifier after a maximum length of 32 characters.

Reserved Words

Statements	Operators	Directives	Other
assert	defined	#assert	const
break	sizeof	#define	forward
case		#else	native
continue		#elseif	new
default		#endif	operator
do		#endinput	public
else		#error	state
exit		#file	static
for		#if	stock
goto		#include	tagof
if		#line	
return		#pragma	
sleep		#section	
switch		#tryinclude	
while		#undef	

There are also dynamically reserved words which vary based upon the content of the user's project. Page names, widget names, image names, and the names given to Meta Refresh objects are also used to reference objects outside of GEMscript, so they pollute the namespace inside GEMscript and you cannot use them for your own identifiers.

Next to reserved words, GEMscript also has several predefined constants. You cannot use the symbol names of the predefined constants for variable or function names.

Constants (literals)

Constants are values, arrays, or strings that are specified literally in an expression.

Integer numeric constants

binary	0b followed by a series of the digits 0 and 1
decimal	a series of digits between 0 and 9
hexadecimal	0x followed by a series of digits between 0 and 9 and the letters a to f

In all number radices, the single quote may be used as a "digit group separator". For decimal numbers, the quote separates 3 digits (thousands separator), for binary numbers, the quote separates 8 digits and for hexadecimal numbers, a quote separates 4 (hexadecimal) digits. Using the single quote as a digit group separator is optional, but if used, the number of digits after the quote must conform to the above specification.

Rational Number Constants

A rational number is a number with a fractional part. A rational number starts with one or more digits, contains a decimal point and has at least one digit following the decimal point. For example, "12.0" and "0.75" are valid rational numbers. Optionally, an exponent may be appended to the rational number; the exponent notation is the letter "e" (lower case) followed by a signed integer numeric constant. For example, "3.12e4" is a valid rational number with an exponent. The single quote may be used as a "thousands separator" (separating 3 digits) in the "whole part" of the number.

These constants are stored in memory according to the IEEE 754 standard of 32-bit floating point.

Character Constants

A single ASCII character surrounded by single quotes is a character constant (for example: 'a', '7', '$'). The single quote that starts a character constant may either be a normal (forward) single quote or a reverse single quote. The terminating quote must always be a forward single quote. Character constants are assumed to be numeric constants.

Escape sequences

'\a'	Audible alarm (beep)
'\b'	Backspace
'\e'	Escape
'\f'	Form feed
'\n'	New-line
'\r'	Carriage Return
'\t'	Horizontal tab
'\v'	Vertical tab
'\\'	\ the escape character
'\''	' single quote
'\"'	" double quote
'\%'	% percent sign
'\ddd;'	character code with decimal code "ddd"
'\xhhh;'	character code with hexadecimal code "hhh"

The semicolon after the \ddd; and \xhhh; codes is optional. Its purpose is to give the escape sequence sequence an explicit termination symbol when it is used in a string constant. The backslash ("\") is the default "escape" character. If desired, you can set a different escape character with the #pragma ctrlchar directive.

String Constants

String constants are assumed to be integer arrays with a size that is sufficient to hold all characters plus a terminating null character (0x00). Each string is stored at a unique position in memory; there is no elimination of duplicate strings. A string literal is a series of zero or more UTF-8 characters surrounded by double quotes. Example:

"The quick brown fox jumped over the lazy dog"

The GEMscript compiler packs as many characters as will fit into each element of the integer array. A character is not addressable as a single unit, though one can index a packed array with {} braces on the byte level. Characters from higher in than 0x7F in the Unicode character set are represented by multiple bytes, but when the string consists of only lower ASCII (0x7F and below), where each character is encoded as only a single byte, this byte indexing with {} braces then doubles as a character access.

Escape sequences may be used within strings. See the section on character constants for a list of escape sequences. There is an alternative syntax for "plain strings". In a plain string, every character is taken as-is and escape sequences are not recognized. Plain strings are convenient to store file/resource names, especially in the case where the escape character is also used as a special character by some host application. The syntax for a plain string is the escape character followed by the string in double quotes. The backslash ("\") is the default "escape" character. You cannot enter escape sequences in a plain string: all characters will be taken literally. plain (unpacked) string constant:

\"C:\all my work\novel.rtf"

In the above example, the occurrences of "\a" and "\n" do not indicate escape sequences, but rather the literal character pairs "\" and "a", and "\" and "n".
Plain packed strings exist as well:

\"C:\all my work\novel.rtf"

Two string literals may be concatenated by inserting with an ellipsis operator (three dots, or "...") between the strings. For example:

"The quick " ... "brown fox"

String concatenation is valid for string literals only —string variables must be concatenated with a library function or run-time code.

Array Constants

A series of comma delimited numeric constants between braces is an array constant. Array constants can be used to initialize array variables with and they can be passed as function arguments. Example:

```
//array a is initialized with an array constant
new int a[] = {192, 168, 0, 1}
```

Symbolic Constants

A source file declares symbolic constants with the const instruction. A single const keyword may declare a list of constants with sequentially incremented values and sharing the same type.

const identifier = constant expression

> Creates a single symbolic constant with the value of the constant expression on the right hand of the assignment operator. The constant can be used at any place where a literal number is valid (for example: in expressions, in array declarations and in directives like "#if" and "#assert").

const type { constant list }

> A list of symbolic names, grouped by braces, may follow the const keyword. The constant list is a series of identifiers separated by commas. The first identifier must be explicitly assigned a (numeric) value. Unless overruled, every subsequent constant has the value of its predecessor plus 1.

> The optional type token that follows the const keyword is used as the default type for every symbol in the constant list. The symbols in the constant list may have an explicit type, which overrules the default type.

A symbolic constant that is defined locally, is valid throughout the block. A local symbolic constant may not have the same name as a variable (local or global), a function, or another constant (local or global).

Predefined Constants

name	defined size	description
cellbits	32	The size of a integer in bits
cellmax	2147483647	The largest valid positive value that a cell can hold
cellmin	-2147483648	The largest valid negative value that a cell can hold.
charbits	8	The size of an ASCII character in bits
charmax	255	The largest valid byte value in a string
charmin	0	The smallest valid byte value in a string
debug	0	The debug level; Reserved for future use.
false	0	This constant has a type of bool.
line		The current line number in the source file.
true	1	This constant has a type of bool.
ucharmax	16777215	The highest value of UTF-8 supported (3 bytes)

Functions

Functions can be called from both inside the script, such as from another GEMscript function, or outside the script, such as from a control widget. Additionally, some functions can be called automatically on certain system events. The mechanism to call a function depends on where you are calling the function from.

The following sections describe how to declare, write, and use functions.

Writing Functions

A function declaration specifies the name of the function and, between parentheses, its formal parameters. A function may also return a value, in which the return type may appear before the function name. If the return type is left off, then an integer type is assumed when a return statement exists. A function declaration must appear outside any other functions, and is accessible to any other GEMscript function along the same page hierarchy (ancestors or children). If a semicolon follows the function declaration (rather than a statement), the declaration denotes a forward declaration of the function. Forward declarations are not required. The return statement sets the function result. For example, function `sum` (see below) has as its result the value of both its arguments added together. The return expression is optional for a function, but one cannot use the value of a function that does not return a value.

```
int sum(int a, int b)
    return a + b;
```

Arguments of a function are (implicitly declared) local variables for that function. The function call determines the values of the arguments. Another example of a complete definition of the function `leapyear` (which returns true for a leap year and false for a non-leap year):

```
int leapyear(y)
    return y % 4 == 0 && y % 100 != 0 || y % 400 == 0;
```

Usually a function contains local variable declarations and consists of a compound statement. In the following example, note the assert statement to guard against negative values for the exponent.

```
/* returns x raised to the power of y */
int power(x, y)
{
    assert y >= 0;
    new r = 1;
    for (new i = 0; i < y; i++)
        r *= x;
    return r;
}
```

A function may contain multiple return statements —one usually does this to quickly exit a function on a parameter error or when it turns out that the function has nothing to do. If a function returns an array, all return statements must specify an array with the same size and the same dimensions.

Function arguments (call-by-value versus call-by-reference)

The `factorial` function in the next program has one parameter which it uses in a loop to calculate the faculty of that number. What deserves attention is that the function modifies its argument.

```
/* Calculation of the factorial of a value */
calculate()
{
    new int v = 123;
    new int f = factorial(v);
}

int factorial(int n)
{
    assert n >= 0;
    new result = 1;
    while (n > 0)
        result *= n--;
    return result;
}
```

Whatever (positive) value that "n" had at the entry of the while loop in function `factorial`, "n" will be zero at the end of the loop. In the case of the `factorial` function, the parameter is passed "by value", so the change of "n" is local to the `factorial` function. In other words, function `calculate` passes "v" as input to function `factorial`, but upon return of `factorial`, "v" still has the same value as before the function call.

Arguments that are not arrays can be either passed by value or by reference. The default is "pass by value". To create a function argument that is passed by reference, prefix the argument name with the character &. Example:

```
swap(&a, &b)
{
    new int temp = b;
     b = a;
     a = temp;
}
```

To pass an array to a function, append a pair of brackets to the argument name. You may optionally indicate the size of the array; doing so improves
error checking of the parser. Example:

```
addvector(int a[], const int b[], int size)
{
    for (new i = 0; i < size; i++)
        a[i] += b[i];
}
```

Arrays are always passed by reference. As a side note, array `b` in the above example does not change in the body of the function. The function argument has been declared as `const` to make this

explicit. In addition to improving error checking, it also allows the GEMscript parser to generate more efficient code. To pass an array of literals to a function, use the same syntax as for array initializers: a literal string or the series of array indices enclosed in braces (the ellipsis for progressive initializers cannot be used). Literal arrays can only have a single dimension.

The following snippet calls addvector to add five to every element of the array "vect":

```
new int vect[3] = [ 1, 2, 3 ];
addvector(vect, [5, 5, 5], 3);
// vect[] now holds the values 6, 7 and 8
```

The invocation of a with the string "Hello world" in the first ubiquitous program is another example of passing a literal array to a function.

Calling functions from within GEMscript

When inserting a function name with its parameters in a statement or expression, the function will get executed in that statement/expression. The statement that refers to the function is the "caller" and the function itself, at that point, is the "callee": the one being called. The standard syntax for calling a function is to write the function's name, followed by a list with all explicitly passed parameters between parentheses. If no parameters are passed, or if the function does not have any, the pair of parentheses behind the function name are still present. For example, to try out the power function, the following program calls it thus:

```
powerTest()
{
    new int base = 2;
    new int pow = 8;
    new int result = power(base, pow);
    InternalRAM.string(0) = append("The number ", itoa(base),
        " raised to the power ", itoa(pow), " is ", itoa(result));
}
```

A function may optionally return a value. The `sum`, `leapyear` and `power` functions all return a value, but the `swap` function does not. Even if a function returns a value, the caller may ignore it. For the situation that the caller ignores the function's return value, there is an alternative syntax to call the function, which is also illustrated by the preceding example program calls the power function. The parentheses around all function arguments are optional if the caller does not use the return value. The syntax without parentheses around the parameter list is called the "procedure call" syntax. You can use it only if:

• the caller does not assign the function's result to a variable and does not use it in an expression, or as the "test expression" of an if statement for example;
• the first parameter does not start with an opening parenthesis;
• the first parameter is on the same line as the function name, unless you use named parameters (see the next section).

For example, the `swap` function from the previous topic could be called in either of these ways:

```
swap(v1, v2)
swap v1, v2
```

As you may observe, the procedure call syntax applies to cases where a function call behaves rather as a statement, like in the calls to `swap` in the preceding example. The syntax is aimed at making such statements appear less cryptic and friendlier to read, but not that the use of the syntax is optional.

As a side note, all parentheses in the example program presented in this section are required: the return value of the calls to `append` and `power` are stored in InternalRAM variables. Even functions that do not require any parameters but still return a value must have an empty set of parenthesis after the function name.

Named parameters versus positional parameters

In the previous examples, the order of parameters of a function call was important, because each parameter is copied to the function argument with the same sequential position. For example, with the function weekday (which uses Zeller's congruence algorithm) defined as below, you would call weekday(12,31,1999) to get the week day of the last day of the preceding century.

```
/* returns the day of the week: 0=Saturday, 1=Sunday, etc. */
int weekday(int month, int day, int year)
{
    if (month <= 2);
    month += 12, --year;
    new int j = year % 100;
    new int e = year / 100;
    return (day + (month+1)*26/10 + j + j/4 + e/4 - 2*e) % 7;
}
```

Date formats vary according to culture and nation. While the format month/day/year is common in the United States of America, European countries often use the day/month/year format, and technical publications sometimes standardize on the year/month/day format (ISO/IEC 8824). In other words, no order of arguments in the weekday function is "logical" or "conventional". That being the case, the alternative way to pass parameters to a function is to use "named parameters", as in the next examples (the three function calls are equivalent):

```
new int wkday1 = weekday( .month = 12, .day = 31, .year = 1999);
new int wkday2 = weekday( .day = 31, .month = 12, .year = 1999);
new int wkday3 = weekday( .year = 1999, .month = 12, .day = 31);
```

With named parameters, a period (".") precedes the name of the function argument. The function argument can be set to any expression that is valid for the argument. The equal sign ("=") does in the case of a named parameter not indicate an assignment; rather it links the expression that follows the equal sign to one of the function arguments.

One may mix positional parameters and named parameters in a function call with the restriction that all positional parameters must precede any named parameters.

Default values of function arguments

A function argument may have a default value. The default value for a function argument must be a constant. To specify a default value, append the equal sign ("=") and the value to the argument name. When the function call specifies an argument placeholder instead of a valid argument, the default value applies. The argument placeholder is the underscore character ("_"). The argument placeholder is only valid for function arguments that have a default value. The rightmost argument placeholders may simply be stripped from the function argument list. For example, if function increment is defined as:

```
increment(&value, incr=1) value += incr;
```

the following function calls are all equivalent:

```
increment(a);
increment(a, _);
increment(a, 1);
```

Default argument values for passed-by-reference arguments are useful to make the input argument optional. For example, if the function divmod is designed to return both the quotient and the remainder of a division operation through its arguments, default values make these arguments optional:

```
divmod(a, b, &quotient=0, &remainder=0)
{
    quotient = a / b;
    remainder = a % b;
}
```

With the preceding definition of function divmod, the following function calls are now all valid:

```
new p, q;
divmod(10, 3, p, q);
divmod(10, 3, p, _);
divmod(10, 3, _, q);
divmod(10, 3, p);
divmod 10, 3, p, q;
```

Default arguments for array arguments are often convenient to set a default string or prompt to a function that receives a string argument. For example:

```
set_error(const string message{}, const string title{} = "Error: ")
{
    InternalRAM.string(0) = title;
    InternalRAM.string(1) = message;
}
```

The next example adds the fields of one array to another array, and by default increments the first three elements of the destination array by one:

```
addvector(a[], const b[] = {1, 1, 1}, size = 3)
```

```
{
    for (new i = 0; i < size; i++)
        a[i] += b[i];
}
```

sizeof operator & default function arguments

A default value of a function argument must be a constant, and its value is determined at the point of the function's declaration. Using the "`sizeof`" operator to set the default value of a function argument is a special case: the calculation of the value of the `sizeof` expression is delayed to the point of the function call and it takes the size of the actual argument rather than that of the formal argument. When the function is used several times in a program, with different arguments, the outcome of the "`sizeof`" expression is potentially different at every call —which means that the "default value" of the function argument may change.

Below is an example program that draws ten random numbers in the range of 0–51 without duplicates. An example for an application for drawing random numbers without duplicates is in card games —those ten numbers could represent the cards for two "hands" in a poker game. The virtues of the algorithm used in this program, invented by Robert W. Floyd, are that it is efficient and unbiased —provided that the pseudo-random number generator is unbiased as well.

```
@load()
{
    new int HandOfCards[10];
    FillRandom(HandOfCards, 52);

    InternalRAM.string(0) = "A draw of 10 numbers from a range of 0
to 51 "
        ..."(inclusive) without duplicates:\n";

    for (new int i = 0; i < sizeof HandOfCards -1; i++)
        InternalRAM.string(0) += append(itoa(HandOfCards[i]), ",
");
    InternalRAM.string(0) += itoa(HandOfCards[(sizeof
HandOfCards)-1]);
}

FillRandom(int Series[], int Range, int Number = sizeof Series)
{
    assert Range >= Number;
    /* cannot select 50 values
     * without duplicates in the
     * range 0..40, for example */
    new int Index = 0;
    for (new int Seq = Range - Number; Seq < Range; Seq++)
    {
        new int Val = random(Seq + 1);
        new int Pos = InSeries(Series, Val, Index);
        if (Pos >= 0)
        {
            Series[Index] = Series[Pos];
            Series[Pos] = Seq;
        }
        else
            Series[Index] = Val;
        Index++;
```

```
        }
}

InSeries(int Series[], int Value, int Top = sizeof Series)
{
        for (new int i = 0; i < Top; i++)
            if (Series[i] == Value)
                 return i;
        return -1
}
```

Function `@load` declares the array `HandOfCards` with a size of ten elements and then calls function `FillRandom` with the purpose that it draws ten positive random numbers below 52. Observe, however, that the only two parameters that `@load` passes into the call to `FillRandom` are the array `HandOfCards`, where the random numbers should be stored, and the upper bound "52". The number of random numbers to draw ("10") is passed implicitly to `FillRandom`.

The definition of function `FillRandom` below `@load` specifies for its third parameter "`int Number = sizeof Series`", where "`Series`" refers to the first parameter of the function. Due to the special case of a "sizeof default value", the default value of the Number argument is not the size of the formal argument `Series`, but that of the actual argument at the point of the function call: `HandOfCards`.

Note that inside function `FillRandom`, asking the "`sizeof`" the function argument Series would (still) evaluate in zero, because the `Series` array is declared with unspecified length. Using `sizeof` as a default value for a function argument is a specific case. If the formal parameter `Series` were declared with an explicit size, as in `Series[10]`, it would be redundant to add a Number argument with the array size of the actual argument, because the parser would then enforce that both formal and actual arguments have the size and dimensions.

Variable arguments

A function that takes a variable number of arguments, uses the "ellipsis" operator ("...") in the function header to denote the position of the first variable argument. The function can access the arguments with the predefined functions numargs, getarg and setarg. Function sum returns the summation of all of its parameters. It uses a variable length parameter list.

```
int sum(...)
{
    new int result = 0;
    for (new int i = 0; i < numargs(); ++i)
        result += getarg(i);
    return result;
}
```

This function could be used in:

```
new v = sum(1, 2, 3, 4, 5);
```

A type may precede the ellipsis to enforce that all subsequent parameters have the same type, but otherwise there is no error checking with a variable argument list and this feature should therefore be used with caution. The functions getarg and setarg assume that the argument is passed "by reference". When using getarg on normal function parameters (instead of variable arguments) one should be cautious of this, as neither the compiler nor the abstract machine can check this. Actual parameters that are passed as part of a "variable argument list" are always passed by reference.

Coercion rules

If the function argument, as per the function definition (or its declaration), is a "value parameter", the caller can pass as a parameter to the function:

- a value, which is passed by value;
- a reference, whose dereferenced value is passed;
- an (indexed) array element, which is a value.

If the function argument is a reference, the caller can pass to the function:

- a value, whose address is passed;
- a reference, which is passed by value because it has the type that the function expects;
- an (indexed) array element, which is a value.

If the function argument is an array, the caller can pass to the function:

- an array with the same dimensions, whose starting address is passed;
- an (indexed) array element, in which case the address of the element is passed.

Recursion

The factorial example function earlier in this chapter used a simple loop. An example function that calculated a number from the Fibonacci series also used a loop and an extra variable to do the trick. These two functions are the most popular routines to illustrate recursive functions, so by implementing these as iterative procedures, you might be inclined to think that GEMscript does not support recursion.

Well, GEMscript does support recursion, but the calculation of factorials and of Fibonacci numbers happen to be good examples of when not to use recursion. Factorial is easier to understand with a loop than it is with recursion. Solving Fibonacci numbers by recursion indeed simplifies the problem, but at the cost of being extremely inefficient: the recursive Fibonacci calculates the same values over and over again.

The program below is an implementation of the famous "Towers of Hanoi" game in a recursive function:

```
/* The Towers of Hanoi, a game solved through recursion */
@load()
{
     //How many disks?
     new int disks = InternalRAM.byte(0);
    InternalRAM.string(0) = "";
    move 1, 3, 2, disks;
    InternalRAM.string(0) += "Done";
}

move(from, to, spare, numdisks)
{
    if (numdisks > 1)
        move from, spare, to, numdisks-1;
    InternalRAM.string(0) += append("Move pillar ", itoa(from), "
to pillar ", itoa(to), ", ");

    if (numdisks > 1)
        move spare, to, from, numdisks-1;
}
```

Forward declarations

For standard functions the GEMscript compiler does not require functions to be declared before their first use.

To create a forward declaration, precede the function name and its parameter list with the keyword `forward`. An alternative way to forwardly declare a function is by typing the function header and terminating it with a semicolon (which follows the closing parenthesis of the parameter list). The full definition of the function, with a non-empty body, is implemented elsewhere in the source file.

Public functions

A GEMscript program must have at least one public function. The public functions provide entry points to the GEMscript program from the Amulet OS. This means for your buttons, sliders, timers, and other widgets to call a GEMscript function, that function must be a public function. The GEMscript virtual machine can also start execution automatically with a certain type of a public function that acts as an event handler.

To make a function public, prefix the function name with the keyword `public`. Below is an example of something you might have on a password page to initialize some variables. The typical nature of an initialization function is to call it once upon page load.

```
public load()
{
     InternalRAM.string(prompt) = "Enter Your Password";
     InternalRAM.string(password) = "";
}
```

Amulet Events with @

Functions whose name starts with the "@" symbol are also public without having to use the keyword `public`. The Amulet OS has the ability to attach GEMscript functions with specific names to corresponding Amulet OS events. All of these special names start with the "@" character. This gives you the ability to write event handlers in GEMscript by just implementing one of these reserved "@" functions. You do not have to write an event handler for every possible event; just the ones you want to use. One such event is generated upon the completion of the page load and it is called `@load`. So an alternative way to write the public function in the previous example is:

```
@load()
{
     InternalRAM.string(prompt) = "Enter Your Password";
     InternalRAM.string(password) = "";
}
```

The "@" character, when used, becomes part of the function name; that is, in the last example, the function is called `@load`. It is also possible to call these functions from other parts of the Amulet OS, such as a button press. You invoke the function with its full name, including the "@" character in this case.

Listing of @ events

You may implement the following event callback functions once on any compiled page. A compiled page is the concatenation of all ancestors of a particular leaf node page. See [Nested Pages and GEMscript](#) for more info.

Event Function Name	When is it called?
@init	Once each time a page is entered, before any object is painted. Causing a paint here will change the drawing order. Usually the @load event is more appropriate.
@load	Once each time a page is entered, after all objects have been painted, but before the first META Refresh or widget Href is executed and before the Frame Buffer is swapped to the newly drawn page (if using DualFrameBuffer)

Calling Functions from outside GEMscript

Other than the special "@" events, a GEMscript function can be invoked from somewhere in your GEMstudio GUI Project, such as a user interacting with a control widget (i.e. pushing a button) or a timer-based trigger. These functions act as interrupt service routines for the various events you are given access to by the Amulet OS. Outside of GEMscript there is a special syntax to invoke a GEMscript function. It starts with the keyword GEMscript, followed by a period, then the name of the GEMscript method you wish to call. Parenthesis are optional.

GEMscript.*functionName*()

This requires the function, *functionName,* to be declared as a [public](#) function. If your function declaration conforms to the [Allman style](#), it will automatically be populated in the autocomplete dropdown lists of the Event Method List or Page Functions Editor accessed with the TAB key. Assuming you have at least one GEMscript function declared, the initial string "GEMscript." will be available from the autocomplete dropdown menu:
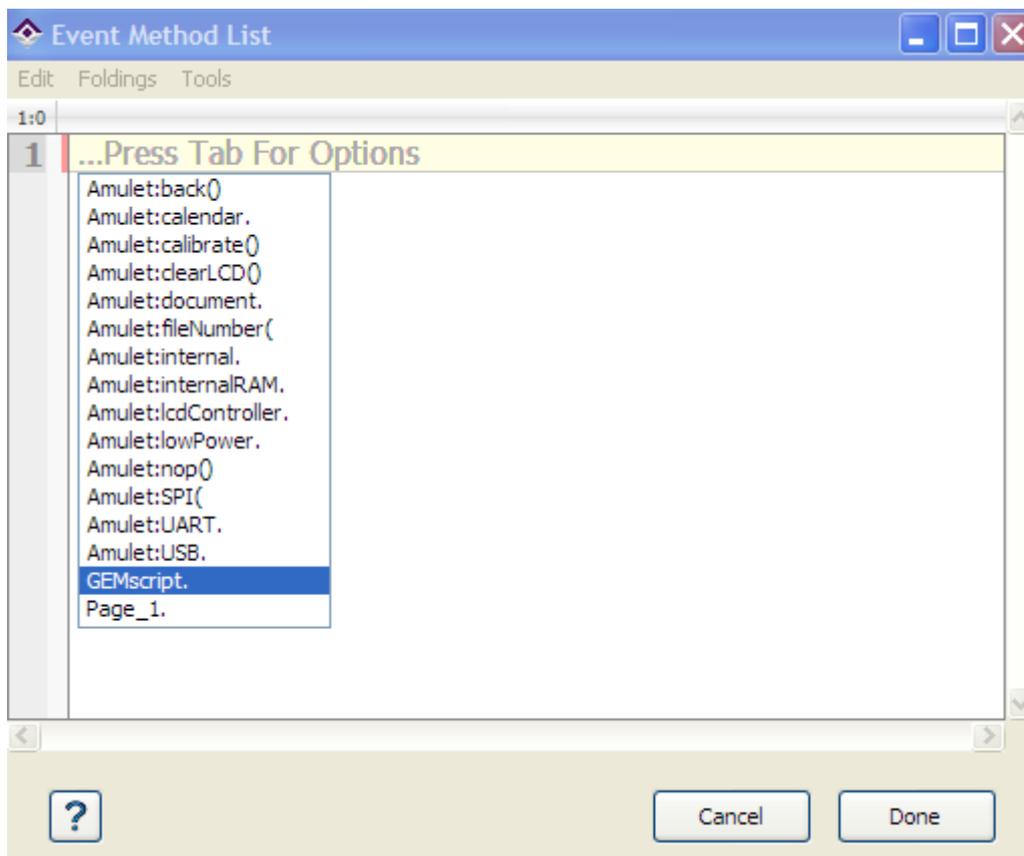
Once you select "GEMscript." you may hit TAB again to see all of the available GEMscript functions for that page.

Using the Autocomplete feature is not necessary in order to call a GEMscript function, but it can help you detect if you have forgotten to make the function public.

Calling Functions on other Pages

It is possible to call functions on other pages, provided those pages are part of the Nested Page hierarchy. The autocomplete feature will reflect this by including all of the public functions on parent pages in the autocomplete dropdown list for the current page. It will not include any public functions from child pages. However, this does not mean that you cannot call functions on child pages. See Nested Pages for more info

Nested Pages and GEMscript

GEMstudio allows pages to be nested. This can be accomplished by simply dragging one page into another. The nested page then contains all of the objects on the parent page, including all of the code in the Page Functions.

GEMscript code in child pages can reference global variables found in the script section of parent pages, but the reverse is not true. A parent page cannot reference a variable declared on a child. This is because a variable must be declared before its first use. When compiling a nested page, only the leaf nodes become actual displayable pages. All of the code from the parent and child pages are concatenated together, starting with the root parent.

On the other hand, functions have a much greater scope because forward references are allowed. In the case of multiple children on one parent page, if you call a function in the parent that exists on one of the child pages, it must exist on ALL child pages. However, in a script you can use compiler switches to disable the code on pages which don't have the function. That might look like this:

```
#if defined Page_Name
// call page-specific function
#else
// do something else
#endif
```

Where Page_Name is the name given to the child page.

For the examples below, consider the following page hierarchy:
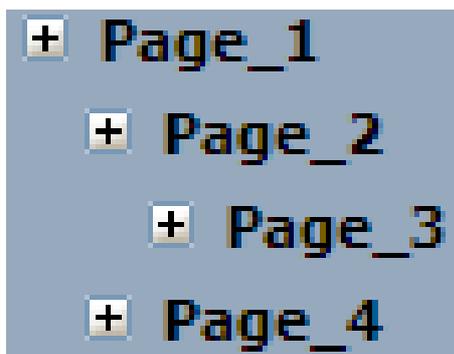
After compiling this project, there are only 2 pages, and their names are Page_3 and Page_4. Page_1 and Page_2 are containers that get concatenated to make the final pages, so calling Page_1.open() would throw an error. The final concatenated script for Page_3 looks like this:

```
<script>
Page_1 code
Page_2 code
Page_3 code
</script>
```

And the final concatenated script for Page_4 looks like this:

```
<script>
Page_1 code
Page_4 code
</script>
```

Now consider this as the complete GEMscript code for Page_1

```
@load()
{
    FunctionA();
}
```

If `FunctionA` is only implemented on Page_2 or Page_3, an error will be thrown when Page_4 is compiled because Page_4 also inherits all of the code from Page_1. There are multiple solutions to this compile error. A brute force method would be to add a blank function with the same signature to Page_4. This gets the job done, but it also wastes memory and time. A more elegant solution is to suppress the compiling of Page_1 code only while compiling Page_4 by the use of compiler directives. The `@load` function could be rewritten like this:

```
@load()
{
    #if defined Page_4
        //do something here
    #else
        FunctionA();
    #endif
}
```

Additionally, from the example page hierarchy and the rules of function and variable scope, we can say that:

- Functions on Page_2 can call functions on Page_3 and Page_1
- Functions on Page_3 can call functions on Page_2 and Page_1
- Functions on Page_4 can call functions on Page_1
- Functions on Page_1 must use compiler directives (like the `#if...#else...#endif` code example above) to call functions that ONLY exist on Page_2, Page_3, or Page_4. Otherwise, a function called on Page_2 or Page_3 must also exist in some form on the compiled Page_4.

- Functions on Page_1 cannot reference variables on any other page.
- Functions on Page_2 can reference global variables on Page_1
- Functions on Page_3 can reference global variables on Page_2 and Page_1
- Functions on Page_4 can reference global variables on Page_1

The preprocessor

The Amulet GEMstudio macro preprocessor allows you to create macros which are used to make textual substitution throughout the project. The macros are defined in a text file with a .macro extension, which is included in your project by choosing a macro file within the Miscellaneous tab in the Project Settings dialog. Almost all text within the project will be scanned by the Amulet macro preprocessor, including GEMscript within the Page Functions editor, and text substitutions will be made at compile time as well as in the WYSIWYG layout editor.

The Amulet macro preprocessor is case sensitive. To be considered an exact match, the macro name must be found in the scanned string and must be surrounded by word separators. The following are considered word separators:

.?!,;:()="

and

start of string, end of string, tabs and spaces.

Note: is NOT considered a space, so if it is desired to use the literal macro name within the project, it can be either preceded or followed by a   which ends up looking on the Amulet LCD as a regular space. Another option is to precede or follow the literal variable name with a &#03; which puts in a space holder that the Amulet doesn't display.

Because the preprocessor scans all the text in your project, it is advised to use macro names that will not show up normally. We suggest starting all of your macro names with an uncommon character, like % or #. Another common practice is to bracket the macro name with %, such as %macro%. This will serve two purposes. One, it makes it easy to see your macros when looking at your code. Two, you won't have to worry about having the preprocessor do a text substitution when you didn't really mean it. The Amulet macro preprocessor is very flexible, though, so you are not required to use any special characters, but care should be taken in the naming of macros.

Enabling the GEM Compiler Preprocessor:

By default, the Amulet GEM Compiler preprocessor is not enabled. You can enable the preprocessor by specifying an include file. To do so, open the project settings, choose on the Miscellaneous tab, and select "Browse...", highlighted in the picture below. The open file dialog will appear. Navigate to your .macro file and select Open.

Defining macros:

All macros must start with `#define`, followed by white space, then the macro name, more white space, then the textual substitution. The macro name is not allowed to have any white space within it. White space is considered either spaces or tabs. Inside the include file, any text to the right of the comment characters // is treated as a comment. It is okay to have a comment on the same line as the macro definition.

As an example, the syntax is as follows:

```
#define macroName textSubstitution
```

Where:
`#define` specifies the creation of a macro.
`macroName` is the name of the macro.
`textSubstitution` is the text which will replace the macro name.

Examples:

```
#define  %hour%     Amulet:UART.byte(0)      // Comment goes here

#define  %getHour   %hour%.value()           // expands to
Amulet:UART.byte(0).value()
```

```
#define   *year        word(3)

#define   +counter    7                              // Comment goes here
```

As you can see from the examples above, it is legal to use macros to define other macros. Forward references are allowed. For example, both of the following are legal:

```
#define   %getCnt    %cnt.value()                    // forward reference to
%cnt is okay!

#define   %cnt       Amulet:InternalRAM.byte(0)
```

as well as

```
#define   %cnt       Amulet:InternalRAM.byte(0)

#define   %getCnt    %cnt.value()
```

Using Macros to initialize InternalRAM:

A powerful feature of the Amulet GEMstudio preprocessor is that it enables you to use defined macros anywhere in the GEMstudio project, including in the initialization of Internal RAM variables. For example, to initialize an InternalRAM byte variable referenced by the macro %counter to a value of 16, you would use the following nomenclature in the InternalRAM initialization file:

```
InternalRAM.byte(%counter) = 16
```

You can also use a macro as the value of an InternalRAM variable as well. For example:

```
InternalRAM.byte(%counter) = %time
```

Directives

#if *constant expression*, **#elseif, #else, #endif**
Portions of a program may be parsed or be ignored depending on certain conditions. The GEMscript parser (compiler or interpreter) generates code only for those portions for which the condition is true. The directive `#if` must be followed by a constant expression. To check whether a variable or constant is defined, use the `defined` operator.

Zero or more `#elseif` directives may follow the initial `#if` directive. These blocks are skipped if any of the preceding `#if` or `#elseif` blocks were parsed (i.e. not skipped). As with the `#if` directive, a constant expression must follow the `#elseif` expression. The `#else` causes the parser to skip all lines up to `#endif` if the preceding `#if` or any of the preceding `#elseif` directives were "true", and then parses these lines if all preceding blocks were skipped. The `#else` directive may be omitted; if present, there may be only be one `#else` associated with each `#if`.

The `#endif` directive terminates a program portion that is parsed conditionally. Conditional directives can be nested and each `#if` directive must be ended by an `#endif` directive.

The most common example is when you have nested pages with one parent page and multiple child pages, and your have code in the Parent page which must do something unique on different child pages. For example:

```
#if defined Child_Page_1
    // call page-specific function
#elseif defined Child_Page_2
    // call page-specific function
#else
    // do something else
#endif
```

GEMscript API

There are many functions available to help you with basic string manipulation and floating point math.

InternalRAM access

InternalRAM variables are the only truly persistent memory. These can be accessed from within GEMscript in two ways. First, InternalRAM commands in the traditional Amulet syntax are available for use. These commands can be found in the GEMstudio Help, under the topic "Amulet Internal RAM", and can be typed just like you would type them anywhere else in the Amulet OS. For example:

```
Amulet:InternalRAM.byte(z).xor(x);
```

GEMscript is a "case sensitive" language: upper and lower case letters are considered to be different letters. One of the few exceptions is InternalRAM access. The keyword InternalRAM is typed differently in numerous examples, and since numerous examples through the years have shown it with varied capitalization, GEMscript is NOT case sensitive on the keyword InternalRAM, the optional "Amulet:" prefix, any names of the data types, or the InternalRAM methods provided by the Amulet OS.

The second and more powerful method to access InternalRAM variables is to bring their values into a script and assign them to GEMscript variables you can use in your code. The members of InternalRAM are specified similar to the syntax above, with a dot after the keyword InternalRAM, then a keyword for the type of data: byte, word, color, or string. The index of the InternalRAM variable immediately follows the type, inside a pair of parenthesis. The index has a range of 0-255 and can be a literal number, variable, or expression that returns an integer type. InternalRAM methods can also be used just like variables, except they do not need to be declared or initialized by GEMscript (however, you can initialize them in GEMstudio). If used as an argument in an expression or anywhere on the right side of the equals sign, this will perform a "get". A "set" will be performed when this is used on the left side of an equals sign (or of any valid assignment operator). For example:

```
new int a = Amulet:InternalRAM.byte(0);
InternalRAM.color(Border_Color) = WHITE; //0x00FFFFFF
InternalRAM.string(i+1) = "Hello World";
```

These can be used for more than simple Get and Set commands. These can also become arguments in expressions, such as a function call parameter or within a math operation. For example:

```
result = Fibonacci(InternalRAM.byte(0));
a = b + InternalRAM.byte(0);
```

The numerical RAM values can even be used as the index argument in InternalRAM calls themselves.

```
InternalRAM.string(InternalRAM.byte(0)) = "Hello World";
```

All assignment operators (such as ++, /=, or *=) also work on the numerical data structures: byte and word. These work as both "Get" and "Set" in one step. You can use these to leverage some extra power from the native Amulet OS in cases where you just want to do a quick modification of InternalRAM. However, there are many cases where you should pull the InternalRAM variable into a new GEMscript variable and then do your calculations, simply because this "Get and Set" still adds overhead to call the Amulet OS methods. The obvious example is the extra overhead accumulated in a loop, but there are others. Consider the following code:

```
while (InternalRAM.byte(0) > 0)
    InternalRAM.byte(0) -= 16;
```

This loop may never exit because InternalRAM values in the Amulet OS are unsigned (but they are cast into signed integers when brought into GEMscript) So, if InternalRAM.byte(0) was odd, the value would never reach zero or less, and the "−=" operation will not clip at zero but instead cause an underflow to wrap the byte back around to a positive number. We can resolve this issue by doing something like this instead:

```
new int a = InternalRAM.byte(0);
while(a > 0)
    a -= 16;
```

Not only will this never be an infinite loop, but it will also execute much faster because GEMscript is not querying the Amulet OS for the InternalRAM value every time through the loop. Bypassing the Amulet OS native operators is not always faster, though.

InternalRAM strings also have an assignment operator, +=, which functions as a native "append" method. Consider the following:

```
Statement1:
{
    InternalRAM.string(0) = "Hello ";
    InternalRAM.string(0) = append(InternalRAM.string(0),"World");
}

Statement2:
{
    InternalRAM.string(0) = "Hello ";
    InternalRAM.string(0) += "World";
}
```

These statements are functionally identical, yet Statement2 executes in considerably less time and uses much less memory because of the overhead of the GEMscript append method. Using the native AmuletOS method is much more efficient, in this scenario.

Hardware Access

Many features of the hardware are currently available in GEMscript.

Communication Ports

The UARTs and USB ports have certain commands exposed in GEMscript. You can either use the built-in Amulet Communications Protocol methods or create your own communications protocol by creating your own state machine. You can then assign the protocol to any of the ports. However, each page of your project is limited to a single protocol, whether it is the Amulet Communications Protocol or the custom created protocol. You have the option of using a different protocol, but it must be utilized on a different page.

Standard Protocol

You can queue the "set command" message of any single external variable (byte, word, color, or string) or an invokeRPC message to go out any of these 4 ports.

1) Set Single External Variable:

Syntax:
Amulet:*port.dataType*(*index*).setValue(*value*)
or
port.dataType(*index*).setValue(*value*)
or
Amulet:*port.dataType*(*index*) = *value*
or
port.dataType(*index*) = *value*

Parameters:
port = USB, uart0, uart1, or uart2 (if applicable)
dataType = byte, word, color, or string
index = variable, expression or static value that evaluates to an integer of value 0x0 - 0xFF.
value = variable, expression or static value that evaluates to the corresponding dataType
       Byte: 0x0 - 0xFF
       Word:0x0 - 0xFFFF
       Color: 0x0 - 0xFFFFFFFF
       String: Null-terminated array{} up to 1k bytes.

Note: for *value* and *index* parameters, values above the specified range will be masked off to fit in this range.

Returns:
1 if there is enough room in the Tx buffer to fit the message, otherwise 0.

Note: Waiting in a loop for communication to finish in order to guarantee that no messages are dropped can result in an unresponsive GUI when the Tx buffer is full.

Examples:
USB.byte(0) = 5
Amulet:UART.word(0) = 1
UART(comm).color(0).setValue(0x000000FF) //Red
Amulet:UART(prog).string(0).setValue("HelloWorld!")

2) Invoke Remote Procedure Call (RPC)

Syntax:
Amulet:*port.*invokeRPC(*index*)
or
*port.*invokeRPC(*index*)

Parameters:
port = USB, uart0, uart1, or uart2 (if applicable)
index = variable, expression or static value that evaluates to an integer of value 0x0 - 0xFF.

Returns:
1 if there is enough room in the Tx buffer to fit the message, otherwise 0.

Note: Waiting in a loop for communication to finish in order to guarantee that no messages are dropped can result in an unresponsive GUI when the Tx buffer is full.

Examples:
Amulet:USB.invokeRPC(0)
UART.invokeRPC(InternalRAM.byte(0))
UART(prog).invokeRPC(GS_Var1)

3) Sending a String of bytes without a response

Syntax:
Amulet:*port.*stringOut*(string)*
or
*port.*stringOut(*string*)

Parameters:
port = USB, uart0, uart1, or uart2 (if applicable)
string = string variable or constant

Returns:
1 if there is enough room in the Tx buffer to fit the message, otherwise 0.

Note: Waiting in a loop for communication to finish in order to guarantee that no messages are dropped can result in an unresponsive GUI when the Tx buffer is full.

Examples:
Amulet:USB.stringOut("Hello World!")

new string str1{} = "Hello World!"
uart1.stringOut(str1)

uart0.stringOut(InternalRAM.string(0))

Custom Protocol

Receiving Bytes

You can define a UART or USB "byte received" interrupt handler. If the function exists, then the Amulet OS will redirect any bytes received to the appropriate function. If these function do not exist, then the OS will retain ownership of the received byte and process it as a standard protocol request or reply. The functions created for the custom protocol can exist on any number of pages. In other words by default the Amulet OS will process the incoming bytes using the Amulet standard protocol, unless a a different custom protocol has been defined.

Syntax:
public @receive_uart0(int *incoming_byte*)
public @receive_uart1(int *incoming_byte*)
public @receive_uart2(int *incoming_byte*)
public @receive_usb(int *incoming_byte*)

Parameters:
incoming_byte is not the required name. You can give this any name you wish, as long as the signature looks the same (one integer parameter)

Returns:
Nothing. Any value you return will be discarded.

Notes:
Defining this function for a given port will redirect all bytes received to that function in the order they are received. If multiple bytes are available, this function will be called one time for each byte. This also means that just defining the function for a particular port will block standard communication on that port. This includes programming messages sent by GEMstudio or any other source, which can be a method of protecting physically exposed ports in the field, but potentially a hassle when developing and reprogramming frequently.

Example:
This example function checks for overflow of a global uart0 receive buffer before calling the state machine to process the buffer.

```
public @receive_uart0(int incoming_byte)
{
    //append new bytes to the end of the buffer
    if(end < RX_BUFFER_SIZE)
    {
        Rx_Buffer{end} = incoming_byte
        end = Process_Buffer()
    }
}
```

Sending Bytes

While you could already send any sequence of bytes out any port with individual sendByte commands in a regular Href, it is often convenient to construct an entire array of bytes and send it out in one command. For example, to calculate the CRC or a message and append it to that message.

Syntax:
sendArray(int *array*{}, int *length*, int *port*)

Parameters:
array: an array containing the bytes to send. Size should be >= length. Required
length: number of bytes to send. Maximum is 0x1000, assuming empty transmit buffer. Required
port: the physical port to send the message on. Required

Notes:
These are for use with the port parameter. You do not need to include these defines in your project.
```
#define USB        4
#define UART0    1+(0<<5)
#define UART1    1+(1<<5)
#define UART2    1+(2<<5)
```

Example:
This example function will setup up a buffer to duplicate the standard CRC protocol command "Get Word Variable Array" and then call sendArray

```
getWord_Array(int address, int count)
{
    new int sendBuffer{6}
    new int CRC
    //setup buffer command
    sendBuffer{0} = 2    //HOST ADDRESS
    sendBuffer{1} = 0x25 //GET_WORD_ARRAY Opcode
    sendBuffer{2} = address & 0xFF
    sendBuffer{3} = count & 0xFF
    CRC = calcCRC(sendBuffer, 6)
    //CRC is added to the buffer in little endian format
    sendBuffer{4} = CRC & 0xFF
    sendBuffer{5} = (CRC >> 8) & 0xFF
    //finally, send the message
    sendArray(sendBuffer,6,UART0)
}
```

Full example

The following code is a full implementation of a very simple state machine that will duplicate the Get Word Variable Array command from the Amulet CRC protocol as an Amulet-As-Master command. The example called by `getWordArray` will send out a request for 3 words, starting at index 0, which looks like this:

0x02 25 00 03 50 56

The reply is an echo of the first four bytes, then the array of data and finally a CRC. An example reply, setting the words to 0x0001, 0x0002, and 0x003 would look like this:

0x02 25 00 03 00 01 00 02 00 03 98 5D

```
<SCRIPT>

//Slave Address configuration:
#define SLAVE_ADDRESS 0x02

//messages/opcodes available on Slave
const {
    GET_WORD_ARRAY = 0x25
    //add more opcodes here, then implement them in Process_Buffer
}

const { //Recieve buffer states enumeration
    SOM   = 0x00,
    SLAVE_OPCODE,
    //states for GET_WORD_ARRAY
    GET_WORD_ARRAY_RESPONSE_INDEX
    GET_WORD_ARRAY_RESPONSE_CNT,
    GET_WORD_ARRAY_RESPONSE_VAL,
    //common CRC states
    CRC_1,
    CRC_2
}

//Global receive buffer
#define RX_BUFFER_SIZE 100
static int Rx_Buffer{RX_BUFFER_SIZE}

//Global state machine variables
static int end = 0



public @receive_uart0(int incoming_byte)
{
    //append new bytes to the end of the buffer
    //incoming_byte<=0x000000FF
    if(end < RX_BUFFER_SIZE)
    {
        Rx_Buffer{end} = incoming_byte
        end = Process_Buffer()
```

```
    }
}
int Process_Buffer()
{
    static int messageLength
    static int Rx_State = SOM
    static int count //for arrays
    switch(Rx_State)
    {
        case SOM: //start of message
        {
            if(Rx_Buffer{end} == SLAVE_ADDRESS)
                Rx_State = SLAVE_OPCODE
            else
                return 0
        }
        case SLAVE_OPCODE:
        {
        //select opcode
            switch(Rx_Buffer{end})
            {
                case GET_WORD_ARRAY:
                {
                    Rx_State = GET_WORD_ARRAY_RESPONSE_INDEX
                    messageLength = 6 //message overhead: slave
addr, opcode, index, count, 2-byte CRC.
                                        //add the count value later,
once we receive it
                }
                //case: //add additional opcodes here
                default:
                {
                    Rx_State = SOM
                    return 0
                }
            }
        }
        case GET_WORD_ARRAY_RESPONSE_INDEX:
        {
            Rx_State++ //GET_WORD_ARRAY_RESPONSE_CNT
        }
        case GET_WORD_ARRAY_RESPONSE_CNT:
        {
            count = Rx_Buffer{end} * 2
            messageLength += count*2
            Rx_State = GET_WORD_ARRAY_RESPONSE_VAL
        }
        case GET_WORD_ARRAY_RESPONSE_VAL:
        {
            //leave the array in the buffer, for now.
            count--
            if(count == 0)
```

```
                Rx_State = CRC_1
        }
        case CRC_1:
        { //two byte CRC, so wait for the next byte to start
processing
            Rx_State++
        }
        case CRC_2:
        {
            if(!calcCRC(Rx_Buffer,messageLength)) //returns 0 if CRC
valid
                GET_WORD_ARRAY_Received();
            Rx_State = SOM
            return 0 //on to next message, reset buffer pointer to
beginning
        }
        default:
            return 0
    } //switch(Rx_State)
    return end + 1
}


#define CRC_SEED   0x0000FFFF
#define CRC_POLY   0x0000A001

int calcCRC(string ptr{}, int cnt)
{
    new int crc = CRC_SEED;     // initialize CRC
    new int i, pos;
    pos = 0;
    while (cnt-- > 0)
    {
        crc = crc ^ ptr{pos++}
        for (i=8; i>0; i--)
        {
           if (crc & 0x0001)
               crc = (crc >> 1) ^ CRC_POLY;
           else
               crc >>= 1;
        }
    }
   return crc;
}

GET_WORD_ARRAY_Received()
{
    //Rx_Buffer contents:
    //slave address - 1 byte
    //opcode - 1 byte
    //starting index - 1 byte
    //count - 1 byte
```

```
    //array of 16-bit words
    new int i
    new int wordCount = Rx_Buffer{3}
    new int startingIndex = Rx_Buffer{2}
    for(i=startingIndex; i<wordCount+startingIndex; i++)
        internalRAM.word(i) = (Rx_Buffer{2*i+4} << 8) +
Rx_Buffer{2*i+5}
}

send_getWordArray(int address, int cnt)
{
    new int sendBuffer{6}
    new int CRC
    //setup buffer command
    sendBuffer{0} = 2    //HOST ADDRESS
    sendBuffer{1} = 0x25 //GET_WORD_ARRAY Opcode
    sendBuffer{2} = address & 0xFF
    sendBuffer{3} = cnt & 0xFF
    CRC = calcCRC(sendBuffer, 4)
    //CRC is added to the buffer in little endian format
    sendBuffer{4} = CRC & 0xFF
    sendBuffer{5} = (CRC >> 8) & 0xFF
    //finally, send the message
    sendArray(sendBuffer,6,UART0)
}
public getWordArray()
{
    send_getWordArray(0,3)
}
</SCRIPT>
```

Notice that states GET_WORD_ARRAY_RESPONSE_INDEX and CRC_1 do exactly the same thing. This is a very common situation in these kinds of state machines, and they could be easily combined into one case statement in the switch.

GPIO

Depending on your hardware, there are up to 8 GPIOs available for you to control from GEMscript.

Setting GPIO to Output High or Low

setGPIO(int port, int value)

Parameters:
port: the particular GPIO signal to drive. The range is 0-7 on the high performance 7 inch capacitive module only. For all other modules the range is 0-1.
Required value: the level to drive the signal at. 0 = low, nonzero = high. Required

Returns:
none

Examples:
//clear all ports
for(new int i = 0; i<8;i++)
   setGPIO(i,0)

Reading a GPIO

under development.

Graphics

The Frame Buffer is a portion of memory dedicated to the display which has traditionally been restricted from direct manipulation by GEMscript. With the introduction of GEMstudio Pro 3.0, there are several graphic commands which allow you to draw simple shapes or get a specific pixel's color from the frame buffer. Note that the top left of the display, respective to its current orientation, is the coordinate 0,0.

drawPixel

Syntax:
drawPixel(int x, int y, int color)

Parameters:
x: the x-coordinate of the pixel. Required
y: the y-coordinate of the pixel. Required
color: the color of the pixel in the format 0xAABBGGRR, where AA=0XFF means Fully Opaque. Required

Returns:
Nothing

drawLine

Syntax:
drawLine(int x1, int y1, int x2, int y2, int color, int thickness=1)

Parameters:
x1: the x-coordinate of the start of the line. Required
y1: the y-coordinate of the start of the line. Required
x2: the x-coordinate of the end of the line. Required
y2: the y-coordinate of the end of the line. Required
color: the color of the pixel in the format 0xAABBGGRR, where AA=0XFF means Fully Opaque. Required
thickness: the thickness of the line. Rage is 1-7. Not required, defaults to 1.

Returns:
Nothing

filledRect

Syntax:
filledRect(int x, int y, int dx, int dy, int color)

Parameters:
x: the x-coordinate of the left of the rectangle. Required
y: the y-coordinate of the top of the rectangle. Required
dx: the width of the rectangle to draw. Required
dy: the height of the rectangle to draw. Required

color: the color of the pixel in the format 0xAABBGGRR, where AA=0XFF means Fully Opaque. Required

Returns:
Nothing

getPixel

Syntax:
getPixel(int x, int y)

Parameters:
x: the x-coordinate of the pixel. Required
y: the y-coordinate of the pixel. Required

Returns:
Nothing

Mass Storage Device

If the Amulet hardware includes multiple flash storage devices, such as eMMC and SD Card, only one of the drives at a time can be assigned as the USB Mass Storage Device. That device will enumerate as a USB Mass Storage Device on assignment. You can switch between the two flash devices as well as prevent either of them from showing up on a computer to protect the installed program from tampering.

setMSDDrive

Syntax:
setMSDDrive(int *value*)

Parameters:
value: the drive to show on the attached computer. Valid range is 0-1

Returns:
Nothing

Notes:
A value of 1 sets the drive to the SD Card, anything else sets it to drive 0, the eMMC.

disableMSD

Syntax:
disableMSD()

Parameters:
None

Returns:
Nothing

Notes:
Disabling the MSD through GEMscript will not prevent users from rebooting the module in Program Mode and bypassing your code completely. To disable this more safely, see the Security tab in the Project Properties in GEMstudio.

enableMSD()

Syntax:
enableMSD()

Parameters:
None

Returns:
Nothing

Notes:

PWM

With the introduction of GEMstudio Pro 3.0, the PWM widget is not the only method to control PWM. Gemscript now allows the control of the PWM through 4 commands. Direct modification is still possible with the PWM widget, and also be used concurrently with the GEMscript PWM commands.

1) Initialize the PWM

The first step to controlling the PWM is initialization. Upon system power up, each PWM port is outputting a logic High. You must initialize each PWM port at some point before you start to use it. This init only has to be done once per power cycle, so you should only every have to call this once for each PWM port you plan to use, typically on some startup page that is only seen once. Here is the syntax:

initPWM(int *channel*)

Parameters:
channel: a variable, expression, or static value that evaluates to an integer from 0 - 2 and determines which hardware port to use

Returns:
nothing

Examples:
initPWM(0)
initPWM(_Buzzer_Channel_)
initPWM()

2) Modify the Period and Pulse Width

Before you start oscillations of the PWM, you probably want to tell the port at what rate and duty cycle it should operate at. Here, the duty cycle is specified as the pulse width, which is in direct proportion of the period. Both are always specified in units of microseconds, which is another difference from the methods of the PWM widget, which are sometimes in milliseconds.. If the pulse width is greater than or equal to the period, the duty cycle is 100%, which is usefull for driving the PWM signal always high. Conversely, if you wanted to drive the line low, a pulse width of 0 will do the job. Here is the syntax:

setPWM(int channel, int period, int pulseWidth)

Parameters:
channel: a variable, expression, or static value that evaluates to an integer from 0 - 2 and determines which hardware port to use.
period: the inverse of the frequency; the duration of the high + low portions of the signal.
pulseWidth: the duration of the high portion of the signal

The range of period is: 10-104000, in microseconds

The range of pulseWidth is: 0-104000, in microseconds

Note: period values smaller than 10 microseconds may result in 50% duty cycles irregardless of pulseWidth

Returns:
nothing

Examples:

setPWM(0,1000,document.MySlider_1) //use a slider's intrinsic value to change the duty cycle.

3) Starting PWM Port Oscillations

After the port is initialized and set to the desired rates, you may start hardware oscillations and it will continue to oscillate at the previously set rates until changed or power is lost. If the port was stopped with the stopPWM command, the PWM may not always be in the high state.

startPWM(int *channel*)

Parameters:
channel: a variable, expression, or static value that evaluates to an integer from 0 - 2 and determines which hardware port to use

Returns:
nothing

Examples:

startPWM(0)
startPWM(_Backlight_Channel_Macro_)

4) Stopping PWM Port Oscillations

You may want to stop oscillations on a PWM port without care to which state the hardware signal comes to a rest at. For example, when driving a piezo buzzer. Here is the Syntax:

stopPWM(int *channel*)

Parameters:
channel: a variable, expression, or static value that evaluates to an integer from 0 - 2 and determines which hardware port to use

Returns:
nothing

Examples:
stopPWM(1)
stopPWM(_Buzzer_Channel_)

Real-Time Clock (calendar)

The calendar object is an easy interface to automatically display the current time using widgets. With 9 different variables available, ranging from year to second, the calendar API for GEMscript has been enhanced slightly. You can use a similar syntax as InternalRAM, such as "calendar.hour", but there are also dedicated "Get" and "Set" methods which allow for greater flexibility.

Note: Optional battery backup of the real-time clock is available only on the high performance 7" capacitive module, MK-070C-HP.

1) GEMscript Exclusive Calendar-Based Methods:

getCalendar(int timeframe)

Parameters:
timeFrame: a variable, expression, or static value that represents unit of time to retrieve. Range is 0-8. See Note.

Returns:
The current value of the given unit of time. The range is dependant on which timeframe is being asked for.

Examples:
```
//Here is a loop to iterate through all values much more
//efficiently than a unique call for each timeFrame.
//Store the year through second into RAM words 0-8
for(new int i = 0; i <=8; i++)
{
    InternalRAM.word(i) = getCalendar(i)
}
```

setCalendar(int timeframe, int value)

Parameters:
timeFrame: a variable, expression, or static value that represents unit of time to retrieve. Range is 0-8. See Note.
value: a variable, expression, or static value for the new calendar setting. The range is dependant on which timeframe is being set.

Returns:
nothing

Examples:
//increment the year
setCalendar(TIMEFRAME_YEAR, getCalendar(TIMEFRAME_YEAR)+1)

Note: The timeFrame parameter has several predefined Macros to help you. Feel free to use these in your own code or make up your own.
```
      TIMEFRAME_YEAR              =0
```

```
TIMEFRAME_MONTH          =1
TIMEFRAME_DAY_OF_MONTH   =2
TIMEFRAME_DAY_OF_WEEK    =3
TIMEFRAME_HOUR           =4
TIMEFRAME_HOUR_MILITARY  =5
TIMEFRAME_AM_PM          =6
TIMEFRAME_MINUTE         =7
TIMEFRAME_SECOND         =8
```

2) Traditional Amulet-Style Calendar Methods

The regular dot syntax which you would use in a widget is also available in GEMscript, with similar features as InternalRAM in GEMscript. For example, the "Amulet:" is optional. You can either get or set these values with the syntax:

Set syntax:
Amulet:calendar.*timeFrame*.setValue(x)
or
calendar.*timeFrame*.setValue(x)

Get syntax:
Amulet:calendar.*timeFrame*.value()
or
calendar.*timeFrame*.value()

Where x is the new value, and *timeFrame* can be either "year", "month", "dayOfMonth", dayOfWeek", "hour", "militaryHour", "am_pm", "minute", or "second". Alternatively, you can skip the value/setValue and use them as arguments or in the left operand of the assignment operator (=).

Combination syntax
Amulet:calendar.*timeFrame*
or
calendar.*timeFrame*

If any one of these is used on the right hand side of an assignment or in an expression, it will return the calendar value. By that same token, if the time frame is followed by an equal sign, then the expression after the equals will be used to set the calendar value.

Examples:

document.YearField = calendar.year
Amulet:calendar.day = document.DaySlider
Amulet:calendar.dayOfWeek.setValue(InternalRAM.byte(0))
new int tempHour = calendar.year.value()

SD Card File I/O

Some Amulet modules are equipped with an SD Card slot which can be used to read and write text files. This File I/O API available through GEMscript allows basic text access initially envisioned to create simple data logging capability. Additional methods exist in the Amulet OS, but are not yet exposed in the GEMscript API. If you have a specific requirement, please contact support@amulettechnologies.com to file a feature request.

Basic Operation

The GEMscript Virtual Machine has access to a file system and retains its own file object so Amulet OS file operations do not interfere, which also limits GEMscript to having a single file open at a time. The function f_chdir(directory) will change the current directory, which starts out in the SD Card root with a logical drive number of 1, i.e. "1:\\". The function f_open(filename) will open a file and leave it open until f_close() is called. All other file access methods access the currently open file in some way. They will return an error if they are called without a currently open file. The examples in the following pages assume that a valid file is already open.

f_chdir

f_chdir   changes current directory to *dir*, a subdirectory of the current, creating each folder if it doesn't already exist.

Syntax:
```
int f_chdir(const string dir{})
```

Parameters:
dir: The directory to create and/or change to. Required

Returns:
0 = no error
non-zero = error

Examples:
```
f_chdir("1:") //go to root
if(f_chdir("Logs")){/*do something*/}
```

Notes:
The default "current directory" is the SD Card root, represented by logical drive number 1.
The double dot ".." in a relative path goes up one level in the directory heirarchy.
The single dot "." in a relative path represents the currently open folder.

The current directory is retained in each volume so that it only affects other tasks which use that volume. That is, Amulet OS tasks such as loading new images and pages from eMMC do not effect the current directory of SD Card. However, if eMMC is bypassed and the GEMstudio project is running from SD Card, then the Amulet OS can change the GEMscript current directory.

Possible error codes are: FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_PATH, FR_INVALID_NAME, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_NOT_ENOUGH_CORE

See File I/O Errors for complete descriptions of each error.

f_close

f_close   closes the currently open file, flushes write buffers

Syntax:
```
int f_close()
```

Parameters:
none

Returns:
0 = no error
non-zero = error

Examples:
```
new int error = f_close()
```

Notes:
Possible error codes are: FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_INVALID_OBJECT, FR_TIMEOUT

See File I/O Errors for complete descriptions of each error.

f_open

f_open   opens the file at *filename* with read and write access, or creates it if it doesn't exist

Syntax:
```
int f_open(const string filename{})
```

Parameters:
filename: The file to open or create, plus an optional directory path. Required

Returns:
0 = no error
non-zero = error

Examples:
```
if(!f_open("Log.txt")) //handle error
```

Notes:
After f_open function succeeds, the current file object is valid. The file object is used for subsequent read/write functions. Open files must be closed prior to shutdown, media removal or re-mount, or the file can be collapsed. To close an open file, use f_close function. Files do not need to be closed between page transitions.

Possible error codes are: FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_NOT_READY, FR_NO_FILE, FR_NO_PATH, FR_INVALID_NAME, FR_DENIED, FR_EXIST, FR_INVALID_OBJECT, FR_WRITE_PROTECTED, FR_INVALID_DRIVE, FR_NOT_ENABLED, FR_NO_FILESYSTEM, FR_TIMEOUT, FR_LOCKED, FR_NOT_ENOUGH_CORE, FR_TOO_MANY_OPEN_FILES

See File I/O Errors for complete descriptions of each error.

f_read

f_read   read data from the currently open file

Syntax:
```
int f_read(string buffer{}, int bytesToRead, int &bytesRead)
```

Parameters:
buffer: The buffer to store incoming read data. Should be at least bytesToRead long. Required
bytesToRead: The number of bytes to attempt to read. Required
bytesRead: A variable for this function to write the number of bytes actually read. Automatically passed by reference. Required.

Returns:
0 = no error
non-zero = error

Examples:
```
new string buffer{100}
new int br
if(!f_read(buffer,100,br))
    document.MyStringField.stringValue = buffer
```

Notes:
bytesRead is automatically passed by reference. You do not need to type a reference operator '&' before the variable when calling f_read

The file read/write pointer of the file object advances by *bytesRead*. After the function succeeded, the variable passed to bytesRead should be checked to detect end of the file. In case of bytesRead is less than bytesToRead, it means the read/write pointer reached end of the file during read operation.

Possible error codes are: FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_INVALID_OBJECT, FR_TIMEOUT

See File I/O Errors for complete descriptions of each error.

f_seek

f_seek   moves the file read/write pointer of the open file. It can also be used to expand the file size (cluster pre-allocation).

Syntax:
```
int f_seek(int offset)
```

Parameters:
offset: Byte offset from top of the file. Required

Returns:
0 = no error
non-zero = error

Examples:
```
f_seek(0) //Seek to beginning of file
f_seek(f_size()) //Seek to end of file
```

Notes:
The offset can be specified in only origin from top of the file. When an offset beyond the file size is specified at write mode, the file size is expanded to the specified offset. The file data in the expanded area is undefined because no data is written to the file. This is suitable to pre-allocate a cluster chain quickly, for fast write operation. After the f_seek function succeeds, the current read/write pointer should be checked with f_tell in order to make sure the read/write pointer has been moved correctly. In case of the current read/write pointer is not the expected value, either of followings has been occured.

•   End of file. The specified *offset* was clipped at end of the file because the file has been opened in read-only mode.
•   Disk full. There is insufficient free space on the volume to expand the file.

Possible error codes are: FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_INVALID_OBJECT, FR_TIMEOUT

See File I/O Errors for complete descriptions of each error.

f_size

f_size   returns the size in bytes of the most recently opened file

Syntax:
```
int f_size()
```

Parameters:
none

Returns:
the size of the file in bytes.

Examples:
```
int size = f_size()
f_seek(size) //seek to end of file.
```

Notes:

f_tell

f_tell gets the current read/write pointer of the most recently opened file.

Syntax:
```
int f_tell()
```

Parameters:
dir: The directory to create and/or change to. Required

Returns:
The bytes offset of the current read/write pointer from the beginning of the file.

Examples:
```
new int pos
f_seek(0)
pos = f_tell() //pos = 0
f_seek(f_size())
pos = f_seek() //pos = f_size()
```

Notes:

f_write

f_write   writes data to a file

Syntax:
```
int f_write(string str{}, int length)
```

Parameters:
str: The buffer to write. Required
length: How many bytes to write.

Returns:
0 = no error
non-zero = error

Examples:
```
f_write("Hello World!",12)
```

Notes:
The read/write pointer of the file object advances by the number of bytes written.
The function can take a time when the volume is full or close to full.

Possible error codes are: FR_OK, FR_DISK_ERR, FR_INT_ERR, FR_INVALID_OBJECT, FR_TIMEOUT

See File I/O Errors for complete descriptions of each error.

File I/O Errors

Error Code	Error Name
	Error Description
0	FR_OK
	The function succeeded.
1	FR_DISK_ERR
	An unrecoverable hard error occured in the lower layer, disk_read, disk_write or disk_ioctl function.

Note that if once this error occured at any operation to an open file, the file object is aborted and all operations to the file except for close will be rejected.

2   FR_INT_ERR

Assertion failed. An insanity is detected in the internal process. One of the following possibilities is suspected.

Work area (file system object, file object or etc...) has been broken by stack overflow or any other tasks. This is the reason in most cases.
There is any error of the FAT structure on the volume.

Note that if once this error occured at any operation to an open file, the file object is aborted and all operations to the file except for close will be rejected.

3   FR_NOT_READY

The storage device cannot work due to a failure of disk_initialize function due to no medium or any other reason.

4   FR_NO_FILE

Could not find the file.

5   FR_NO_PATH

Could not find the path.

6   FR_INVALID_NAME

The given string is invalid as the path name.

7   FR_DENIED

The required access was denied due to one of the following reasons:

Write mode open against the read-only file.
Deleting the read-only file or directory.
Deleting the non-empty directory or current directory.
Reading the file opened without a read flag.
Any modification to the file opened without a write flag.
Could not create the file or directory due to the directory table is full.
Could not create the directory due to the volume is full.

8   FR_EXIST

Name collision. Any object that has the same name is already existing.

9   FR_INVALID_OBJECT

The file/directory object is invalid or a null pointer is given. There are some reasons as follows:

It has been closed, it is not opened or it can be collapsed.
It has been invalidated by a volume mount process. All open objects of the volume are invalidated as well.
The corresponding physical drive is not ready due to a media removal.

10   FR_WRITE_PROTECTED

Any write mode operation against the write-protected media.

11   FR_INVALID_DRIVE

Invalid drive number is specified in the path name. A null pointer is given as the path name.

12   FR_NOT_ENABLED

Work area for the logical drive has not been registered.

13   FR_NO_FILESYSTEM

There is no valid FAT volume on the drive.

14   FR_MKFS_ABORTED

The f_mkfs function aborted before start in format due to a reason as follows:

The disk/partition size is too small.
Not allowable cluster size for this disk. This can occurr when number of clusters gets near the boundaries of FAT sub-types.
There is no partition related to the logical drive.

15   FR_TIMEOUT

The function was canceled due to a timeout of thread-safe control.

16   FR_LOCKED

The operation to the object was rejected by file sharing control.

17   FR_NOT_ENOUGH_CORE

Not enough memory for the operation. There is one of the following reasons:

Could not allocate a memory for LFN working buffer.
Size of the given buffer is insufficient for the size required.

18   FR_TOO_MANY_OPEN_FILES

Number of open objects has been reached maximum value and no more object can be opened.

19   FR_INVALID_PARAMETER

The given parameter is invalid or there is any inconsistent.

SPI

There are up to 3 SPI Chip selects which share a MISO, MOSI, and Clock signals, which can be controlled from GEMscript. The SPI interface is comprised of two commands.

Pulling Chip Selects Low with SPI_Start

Syntax:
SPI_Start(int *port*)

Parameters:
port: the Chip Select to activate. Range is 1-3. Required

Returns:
nothing

Reading and Writing Data With SPI_Out

Syntax:
int SPI_Out(int *byte*, int *lastXfer*)

Parameters:
byte: the 8-bit data to write. Required
lastXfer: determines whether or not to deassert the chip select when done with this byte. 0 = no, nonzero = yes. Required

Returns:
The data clocked in on MISO. Only valid if slave was driving MISO at the time.

Notes:
To read SPI data, configure the slave with SPI_Out commands and ignore the return value, and then send Dummy data to read the value

Examples:

```
#define READ_FLAG 0x80
#define DUMMY 0xFF
readRegister(int register)
{
   SPI_Start(1) //assert CS1
   SPI_Out(register | READ_FLAG,0) //shift out Address with READ flag
   return SPI_Out(DUMMY,1) //shift in data, deassert CS when done
}
writeRegister(int register, int value)
{
   SPI_Start(1) //assert CS1
   SPI_Out(register,0) //shift out Address with WRITE flag
   SPI_Out(value,1) //shift out value and deassert CS when done
}
```

Touchpanel

The GEMscript touchpanel interface is abstracted through the Amulet OS's touchpanel drivers to provide you with a pixel coordinate of the current touch (or lack thereof). This pixel coordinate is particularly convenient if you want to do something like track an image under your finger but you don't want to be restricted by the current set of widgets.

PenX

Syntax:
PenX(int *touchNumber* = 0)

Parameters:
touchNumber: the number of the touch to query. Non-zero only valid for multi-point capacitive touchpanels. Not required. Defaults to 0 (first touch)

Returns:
The pixel x-coordinate of the current touch, or 0xFFFF8000 specifies no touch.

Notes:
Small negative values are common, especially if they touchpanel is physically larger than the display (off screen soft-buttons). This makes common error codes like -1 impossible to use here, so a large negative number was used for the "no touch" code.

PenY

Syntax:
PenY(int *touchNumber* = 0)

Parameters:
touchNumber: the number of the touch to query. Non-zero only valid for multi-point capacitive touchpanels. Not required. Defaults to 0 (first touch)

Returns:
The pixel y-coordinate of the current touch, or 0xFFFF8000 specifies no touch.

Notes:
Small negative values are common, especially if the touchpanel is physically larger than the display (off screen soft-buttons). This makes common error codes like -1 impossible to use here, so a large negative number was used for the "no touch" code.

Widgets

IWC commands

Inter-Widget Communication allows one Amulet widget to invoke the methods of another Amulet widget, but these are also accessible from within GEMscript. Appendix C lists all of the commands available for each type of widget. Similar to InternalRAM access, the "Amulet:" is optional. The keyword "document" is followed by a period, the widget Name, another period, and the IWC method you wish to call, along with any parameters it takes inside parenthesis:

```
Amulet:document.WidgetName.method(x);
OR
document.WidgetName.method(x);
```

The syntax is nearly identical between GEMscript and the rest of GEMstudio, but there are a few important things to note.

1) The Widget name is CASE SENSITIVE, while "Amulet:document" as well as the method name are NOT CASE SENSITIVE.

2) There will be no State Machine set up by use of commas and semicolons.

3) Some IWCs may push additional commands on the Amulet OS command buffer, but these additional commands will not be executed until your script gives control back to the Amulet OS.

> For example, consider the method `forceUpdate` used on a view widget, such as a BarGraph. View widgets operate on a timer counting down to zero. Once the Amulet OS detects the counter hit zero, it will query the HREF variable and determine if the new value is different from the old value. If that is true it will do the redraw. Finally it resets the count to the number specified in the widget's UpdateRate. The `forceUpdate` method simply sets this timer counter to zero and lets the Amulet OS handle querying the HREF and determining if a redraw needs to occur, so a widget will not redraw until the Amulet OS regains control. Normally, this is not of any consequence. However, this becomes critical when draw order is important because objects are layered on top of each other.

> A more elegant solution than calling `forceUpdate` on a widget is to set the [intrinsic value](#) of that widget directly in GEMscript. The difference between this and letting the Amulet OS poll and decide to redraw is that this does not schedule the redraw. It executes exactly when you tell it to. This topic is covered in more detail in the next topic.

Intrinsic Values

The intrinsic value of a widget is the internal value of that widget. In the case of Control widgets like a slider, this is the value that the widget's HREF passes to commands that use the word "intrinsicValue" as an argument. In the case of a View widget, this is the actual data being displayed, before processing such as clipping to be within Min/Max parameters.

For example, a BarGraph widget's HREF may point to an InternalRAM byte variable, but the Amulet OS actually copies that InternalRAM Byte into the BarGraph's personal intrinsic value memory. Additionally, because of Min/Max parameters, the BarGraph drawing output might only care about values from 0-100. If the HREF function returned a value of 300, when you query the intrinsic value of this BarGraph in GEMscript, it will return 300 even though 0-100 are the only values that have any graphical effect (101+ would look just like 100).

Now lets take a look at how you can set and retrieve these values.

```
Amulet:document.widget_name
```
OR
```
document.widget_name
```

The syntax is borrowed from the Amulet OS's "dot syntax", but we've shortened it here for ease of use. If this expression exists on the left side of an assignment, it will call setValue. If the expression exists on the right hand side of an assignment, or as an argument on the left side, then it will perform the value() method to get the widget's intrinsic value. You could explicitly call setValue(x) or value() to provide the same functionality, but this is not needed.

Examples:

```
var_A = document.MySlider;
document.MyNumericField = var_A;
```

In some cases, the intrinsic value is a string type. There is a special syntax for these.

```
Amulet:document.widget_name.stringvalue
```
OR
```
document.widget_name.stringvalue
```

Example:

```
new string str1{100};
str1 = document.MyStringField.stringvalue;

new string str2{} = "Hello World";
document.MyStringField.stringvalue = str2;
```

Floating Point Library

Declaration:

```
new float varName
```
or
```
new float varName = float literal or float expression
```
or
```
new float varName = integer literal or expression
```

Parameters:
new: start of any variable declaration. Required
float: type definition. Tells compiler which methods may use this variable. Required
varName: arbitrary name of the variable. Required
float literal: a constant number containing a decimal point, such as 3.14159, optionally with exponent notation such as 2.99e8
float expression: an expression that evaluates to a float type
integer literal: a constant number without a decimal point, such as 255, 0xFF, or 0b100100. This will get converted to a float at runtime.
integer expression: an expression that evaluates to an int type. This will get converted to a float at runtime.

Valid examples:
```
new float f1;
new float f2 = 1.234;
new float f3 = 5.678e9;
new float f4 = 10;
new float f6 = strtofloat("12.3");  //see conversion methods below for
```
[strtofloat()](#)

// Converting an integer variable into a float:
```
new int a = 10;
new float f7 = float(a);  //see conversion methods below for float()
```

Invalid examples:
```
new float f8 = "3.14";  //cannot initialize directly from string literal.
```

An overloaded assignment operator is implemented to automatically coerce integer values on the right hand to a floating point format on the left hand. That is, the lines:
```
new int a = 10;
new float b = a;
```
are equivalent to:
```
new a = 10;
new float b = float(a);
```

Notes:
Only lower ASCII characters A-Z, a-z, 0-9, and _ are allowed in the variable name

The internal memory layout of the float is the 32-bit IEEE 754 standard.

Conversion

The following methods help convert to and/or from Floating Point types.

float

float	**converts an integer to a floating point value**

Syntax:
```
float float(int intVal)
```

Parameters:
intVal: the integer to convert. Required

Returns:
the floating point representation of the integer *intVal*

Usage:
```
//to explicitly convert an integer for use as a float argument:
new float a, b;
new int c = 4;
new int d = 6;
a = floatpower(float(c),0.5); // a = 2
b = float(d/c);                 // b = 1.0 because the math was done
with integers, before the float conversion.
b = float(c) / float(d);     // b = 1.5
b = float(c) / d;             // b = 1.5 because the / operator is
overloaded to convert either side to a float.
```

Notes:
This function is automatically used by all overloaded operators so that you can used mixed types in your calculations.
Arguments for the more advanced math functions (the ones without common operators) will require explicit conversion using this method.

strtofloat

strtofloat   **converts a string to a floating point value**

Syntax:
```
float strtofloat(const string source{})
```

Parameters:
source: The string to be converted. Required

Returns:
The floating point value of the string

Usage:
```
new string a{} = "3.14159";
new float b = strtofloat(a);
```

Notes:

floattostr

floattostr   **converts a floating point value to a string**

Syntax:

```
floattostr(float floatVal, string dest{}, int length)
```

Parameters:

floatVal: the value to be converted. Required
dest: the destination buffer. Required
length: maximum length of the desired string. Required

Usage:

```
new float f = 3.14159;
new string c{10};
floattostr(f,c,10);
```

Notes:

floatround

floatround   **rounds a floating point number and converts it to an integer**

Syntax:
```
int floatround(float value, int method=floatround_round);
```

Parameters:
value: the value to round. Required
method: the style of rounding. Optional. Defaults to nearest integer

Returns:
the integer representation of *value*

Usage:
```
new float a = -1.234;
new int b;
b = floatround(a);                       // b = -1
b = floatround(a, floatround_floor);  // b = -2
b = floatround(a, floatround_tozero); // b = -1
```

Notes:
the rounding methods are described by the following:

```
const floatround_method: {
floatround_round  = 0, // standard rounding towards nearest
integer
floatround_floor  = 1, // round downwards
floatround_ceil   = 2, // round upwards
floatround_tozero = 3  // truncation
}
```

floatfract

floatfract   **returns the fractional part of the float**

Syntax:
```
float floatfract(float value)
```

Parameters:
value: the floating point value to convert. Required

Returns:
the fractional part of the float

Usage:
```
new float a = 54.321;
new float b = floatfract(a); // b = 0.321
```

Notes:
This is similar to performing modulo operation % 1.0

Math

The following methods provide additional functionality to floating point arithmetic.

floatpower

floatpower   **returns the value raised to the power of the exponent**

Syntax:
```
float floatpower(float value, float exponent)
```

Parameters:
value: the floating point value to be raised. Required
exponent: the floating point power to raise the value by. Required

Returns:
$value^{exponent}$

Usage:
```
new float a = floatpower(4, 2);  // a = 16
```

Notes:
`floatpower(value, 0.5)` is the same as `floatsqroot(value)`

floatsqroot

floatsqroot   **returns the square root of the value**

Syntax:
```
float floatsqroot(float value)
```

Parameters:
value: the value to take the square root of. Required.

Returns:
√*(value)*

Usage:
```
new float a = floatsqroot(4.0); // a = 2.0
```

Notes:
This is the same as `floatpower(value, 0.5)`

floatlog

floatlog   **returns the logorithm of the value**

Syntax:
```
float floatlog(float value, float base=10.0)
```

Parameters:
value: the number to take the logorithm of. Required.
base: the base used in the logorithm calculation. Optional. Defaults to 10.0

Returns:
$\log_{base}(value)$

Usage:
```
new float a = floatlog(100.0);      // a = 2.0
new float b = floatlog(125.0,5.0); // b = 3.0
```

Notes:

floatcos

floatcos   **returns the cosine of the angle**

Syntax:
```
float floatcos(float angle, int mode=radian)
```

Parameters:
angle: the value to calculate the cosine of. Required.
mode: the units that angle is specified in. Optional. Defaults to radians. See notes.

Returns:
the cosine of *angle*

Usage:
```
new float a = floatcos(3.14159265359); // a = -1
new float b = floatcos(90.0,degrees);  // b = 0
```

Notes:
The units the angle is specified in is defined by the following:

```
const anglemode: {
radian = 0,
degrees = 1,
grades = 2
}
```

floatsin

floatsin   **returns the cosine of the angle**

Syntax:
```
float floatsin(float angle, int mode=radian)
```

Parameters:
angle: the value to calculate the sine of. Required.
mode: the units that angle is specified in. Optional. Defaults to radians. See notes.

Returns:
the sine of *angle*

Usage:
```
new float a = floatsin(3.14159265359); // a = 0
new float b = floatsin(90.0,degrees);  // b = 1
```

Notes:
The units the angle is specified in is defined by the following:

```
const anglemode: {
radian = 0,
degrees = 1,
grades = 2
}
```

floattan

floattan   **returns the cosine of the angle**

Syntax:
```
float floattan(float angle, int mode=radian)
```

Parameters:
angle: the value to calculate the tangent of. Required.
mode: the units that angle is specified in. Optional. Defaults to radians. See notes.

Returns:
the tangent of *angle*

Usage:
```
new float a = floattan(3.14159265359);  // a = 0
new float b = floattan(90.0,degrees);   // b = 1
```

Notes:
The units the angle is specified in is defined by the following:

```
const anglemode: {
radian = 0,
degrees = 1,
grades = 2
}
```

floatabs

floatabs   **returns the absolute value of the given value**

Syntax:
```
float floatabs(float value)
```

Parameters:
value: the floating point number to take the absolute value of.

Returns:
if >0, *value*
otherwise (*-value*)

Usage:
```
new float a = floatabs(-1.0);  // a = 1.0
```

Notes:

Supported Math Operators

The following operators can be used with float variables:
Common math operators:

+   -   *   /   %

Assignment operators:

=   ++   --   +=   -=   *=   /=   %=

Compare operators:

==   !=   >   <   >=   <=   !

These operators can also work with mixed types on either side of the operator, regardless of if the function of that operator is commutative or not. Integer types will be automatically converted to floating point prior to evaluating the expression. For example:

new int a = 10;
new float b = 0.5;
b+=a; // this expands to b = b + float(a)

The following operators are not compatible with float variables:

^   ^=   &   &= | |=

String Library

Declaration:

```
new string varName{} = string literal
or
new string varName{numBytes}
or
new string varName{numBytes} = string literal
```

Parameters:
new: start of any variable declaration, required
string: type definition, tells compiler which methods may use this variable, required
varName: arbitrary name of the variable, required
{numBytes}: denotes length of string buffer in bytes. GEMscript rounds all arrays up to multiples of 4 bytes. For example, {5} will actually allocate 8 bytes. Required
= string literal: initialization is optional, but any assignment must be to a string literal encapsulated by double quotes, i.e. "this is a string literal". Uninitialized data defaults to zero.

Valid examples:
```
new string strVar1{} = "Hello World";
new string strVar2{100} = "Hello World";
new string strVar3{GEMscriptStringBufferSize};
```

Invalid examples:
```
new string strVar4{};             // unknown size of string
new string strVar5{3} = 4;        // cannot set array to a value
new string strVar6 = strVar1;     // string initializers must be
constants.
new int notArray = "Hello World"; // only an array can hold a
string.
```

Notes:
• Only lower ASCII characters A-Z, a-z, 0-9, and _ are allowed in the variable name.
• The {} should not be appended to the variable when calling any function, nor should the reference operator '&'.   The brackets in the method signature are merely to denote that the compiler expects an array variable type.

InternalRAM.string Input and Output

There is a special syntax to reference the InternalRAM of the Amulet OS, which is similar to how you would type a reference to the same variables outside of a script, such as in an HREF of a stringField widget. The following sections describe how to set and get the InternalRAM strings.

Input from InternalRAM

InternalRAM.string   **returns a copy of an InternalRAM String variable**

Syntax:
```
string variable = InternalRAM.string(int ramIndex)
```

Parameters:
variable: destination array variable with some minimum size (see note)
ramIndex: a numerical value, variable, or expression

Usage:
```
new string IR_String1{GEMscriptStringBufferSize};   // we need to
create a buffer for the string
IR_String1 = InternalRAM.string(0);
```

Invalid Usage:
```
new string IR_String2{GEMscriptStringBufferSize} =
InternalRAM.string(0); // strings initializers must be constants
```

Notes:
The returned array has a default size defined by the macro
GEMscriptStringBufferSize, so any buffers you define should be at least this big or
the compiler will throw an error.

Output to InternalRAM

InternalRAM.string   **sets an InternalRAM String variable**

Syntax:
```
InternalRAM.string(int ramIndex) = string stringValue
```

Parameters:
ramIndex: The InternalRAM index to set. Can be a numerical value, variable, or expression
stringValue: The string to set the InternalRAM variable to. Can be a string literal, variable, or expression

Usage:
```
InternalRAM.string(0) = "Hello World";
InternalRAM.string(index) = strVar1;
InternalRAM.string(index+1) = append("Hello ", "World");
InternalRAM.string(InternalRAM.byte(0)) = InternalRAM.string(255);
```

On top of the '=' assignment operator, there is also an append "+=" assignment operator which does not clear the source string:
```
InternalRAM.string(0) = "Hello ";
InternalRAM.string(0) += "World"; //string 0 now contains "Hello World"
```

Notes:

The syntax is also valid:
```
InternalRAM.string(0) = append(InternalRAM.string(0), "appendedString");
InternalRAM.string(0) += "appendedString";
```
Notice that the first is a 3 step process. From inside out: query, concatenate, set. The second is far more efficient because it will perform the same function in one "set-append" assignment operator. This is not more efficient when you want to do multiple operations on a string. In that case, it is usually better to copy the InternalRAM string into a local buffer, do your manipulation, then copy it back out to InternalRAM.

Accessing Elements in a string

Element Access:

{}   **returns the byte at zero-based offset** *n*

Syntax:
```
stringVar{n}
```

Parameters:
stringVar: the source string
n: the offset into the source string

Usage:
```
new string str1{16} = "Hello World";
new int chr = str1{0};              // variable chr now contains
the value 0x48 ('H')
str1{11} = '!';                     // variable str1 now
contains "Hello World!"
```

Notes:
When the string content is lower ASCII only, this is a direct character access. Otherwise it is a byte access of UTF-8 encoding that has the most significant byte in the encoded character first. For UTF-8 character manipulation, use String API methods where the method name does not end with 'b' (i.e. strins vs. strinsb). In the present release only the methods ending in b are implemented.

Properties

There are two main properties of a string: the size of the buffer it resides in, and the length of the string itself. The latter is defined by how many bytes from the beginning until a "null" byte with a value of 0 is encountered.

strlenb

strlenb   **returns the length of the string in bytes**

Syntax:
```
int strlenb(const string source{})
```

Parameters:
source: The source string

Usage:
```
new string strVar{100} = "Hello World";
new int strLen = strlenb(strVar);        // strLen is now set to
11
```

sizeof

sizeof   returns the number of 4-byte chunks allocated to the string buffer

Syntax:
```
int sizeof(varName)
```

Parameters:
varName: the name of the variable. This can be any data type.

Usage:
```
new int a,b;
new string c{5};
new int d[5];
new float e;
a = sizeof(b); // a = 1
a = sizeof(c); // a = 2
a = sizeof(d); // a = 5
a = sizeof(e); // a = 1
```

Notes:
Notice the string has a length of 5, but sizeof returns a size of two 4-byte chunks.
This is because the compiler will round up all string buffers to multiples of 4 bytes.

Conversions

The following methods provide conversions to and from string arrays.

strtoval

strtoval   **returns the signed 32-bit integer value of the string**

Syntax:
```
int strtoval(const string source{})
```

Parameters:
string: the source array to be converted

Usage:
```
new string numberString = "1234";
new int numberVal = strtoval(numberString); //numberVal now
contains the numerical value 1234
```

Notes:
For floating point numbers, there is a strtofloat method

valtostr

valtostr   **converts value to a string and saves the value to the dest buffer**

Syntax:
```
valtostr(string dest{}, int value)
```

Parameters:
dest: the destination string buffer where the converted value will be saved
value: the 32-bit number to be converted.

Usage:
```
new string buffer{10};
new intVar = 5;
valtostr(buffer, 10 * -2 + intVar); // buffer contains "-25"
```

itoa

itoa   **returns a string converted from *value* using valtostr**

Syntax:
```
string itoa(int value)
```

Parameters:
value: the 32-bit number to be converted.

Returns:
A new string buffer containing the converted string.

Usage:
```
new string buffer{GEMscriptStringBufferSize};
buffer = itoa(1234);
InternalRAM.string(0) = itoa(1234);
new int i = strtoval(itoa(1234));
```

Notes:
The returned array has a default size defined by the macro
GEMscriptStringBufferSize, so any buffers you define should be at least this big or
the compiler will throw an error.

String Modifications

The following methods modify a source string. Some return a new string and others use a destination buffer that is passed.

append

append   **returns a new string that is the concatenation of up to 10 source strings**

Syntax:
```
string append(string source1{}, string source2{}="", ...string
source10{}="")
```

Parameters:
string1: First source string; Can be a literal string, string variable, or string expression. Required
string2-10: Optional source strings appended to *string1* in the same order they appear on the argument list.

Usage:
```
new string a{GEMscriptStringBufferSize};
a = append("Hello ", "World"); // variable a contains the string
"Hello World"
new string b{100};
b = a.append("!!!");           // variable b contains the string
"Hello World!!!"
b = append(a, "!!!");          // exactly the same function as the
previous line
```

Notes:
The returned array has a default size defined by the macro
GEMscriptStringBufferSize, so any buffers you define should be at least this big or the compiler will throw an error.

strins

strins   **inserts a substring into a source string using a character index**

Reserved. Not implemented yet.

strinsb

strinsb   **inserts a substring into a source string using a byte index**

Syntax:
```
int strinsb(string string{}, const string substr{}, int index, int
maxlength=sizeof string)
```

Parameters:
string: The string to be modified; Required
substr: The string to be inserted; Required
index: The byte offset to insert *substr*; Required
maxlength: Limit on how long *string* can become, counted in 32-bit increments

Returns:
true if successful. Otherwise false, for example if *index* is past the terminating null character.

Usage:
```
new string a{15} = "Hello !";
strinsb(a,"World", 6);          // variable a now contains the
string "Hello World!"
```

Notes:
If operating on strings containing characters outside of the lower ASCII range, `strinsb` can actually insert in between two bytes of a multi-byte character, which corrupts the UTF-8 encoding. The function strins is safer because it operates on UTF-8 characters.

strdel

strdel   **removes bytes from a string by offset**

Reserved. Not implemented yet.

strdelb

strdelb	**removes bytes from a string by offset**

Syntax:
```
int strdelb(string source{}, int start, int end)
```

Parameters:
source: the string to modify. Required.
start: the index to start deleting. Required.
end: the index to end deleting. This index is NOT deleted. Required.

Returns:
true if successful. Otherwise false, for example if *start* is past the terminating null character, or if (*end* - *start* <= 0)

Usage:
```
new string a{} = "Hellooooo World!";
strdelb(a,5, 9);                      // variable a now contains
the string "Hello World!"
```

Notes:
If operating on strings containing characters outside of the lower ASCII range, `strdelb` can actually delete single bytes of a multi-byte character, which corrupts the UTF-8 encoding. The function strdel is safer because it operates on whole UTF-8 characters.

strcopy

strcopy	**copies one string to another string**

Syntax:
```
strcopy(string dest{}, const string source{}, maxlength=sizeof
dest)
```

Parameters:
dest: the destination buffer. Required
source: the string to copy. Required
maxlength: maximum number of 32-bit chunks to copy over. Optional. Defaults to the max length of the destination buffer.

Usage:
```
new string a{100} = "Hello World";
new string b{15};
strcopy(b,a);    // variable b now contains the string "Hello
World"

b = a;           // Throws error 47: array sizes/definitions do not
match, or destination array is too small
```

Notes:
Most of the time a simple "b = a" will work just fine. This method is useful when the destination buffer is smaller than the source, saving you the hassle of copying each byte over manually.

strcat

strcat

Syntax:
```
strcat(string dest{}, const string source{}, int maxlength=sizeof
dest)
```

Parameters:
dest: the destination buffer. Required
source: the string to append to *dest*. Required
maxlength: maximum number of 32-bit chunks to copy over. Optional. Defaults to
the max length of the destination buffer.

Usage:
```
new string a{15} = "Hello";
new string b{} = " World";
strcat(a,b);                   // variable a now contains the string
"Hello World"
```

Notes:
Similar to the append function, but the destination buffer must be passed, so it
does not return anything.

strpad

strpad   **pads the beginning of a string**

Syntax:
```
int strpad(string dest{}, int length, int chr = ' ')
```

Parameters:
dest: the string to pad. Required
length: the length of the string after padding to the left. Required
chr: the character to pad with. Optional. Defaults to the space character (0x20)

Returns:
the count of bytes padded

Usage:
```
new string a{20} = "Hello World";
strpad(a,16);                        // variable a now contains the
string "     Hello World"
new string b{9} = "Pad me";
strpad(b, 9, 'P');                   // variable b now contains the
string "PPPPad me"
```

Notes:
The single quote notation returns the numerical value of the ASCII character between the single quotes.

strrpad

strrpad   **pads the end of a string**

Syntax:
```
int strrpad(string dest{}, int length, int chr = ' ')
```

Parameters:
dest: the string to pad. Required
length: the length of the string after padding to the right. Required
chr: the character to pad with. Optional. Defaults to the space character (0x20)

Returns:
the count of bytes padded

Usage:
```
new string a{20} = "Hello World";
strrpad(a,16);              // variable a now contains the string
"Hello World     "
new string b{8} = "Pad me";
strrpad(b, 9, 'P');        // variable b now contains the string
"Pad mePP" because the max size of the buffer was 8
```

Notes:
The single quote notation returns the numerical value of the ASCII character between the single quotes.

strrtrim

strrtrim   **removes any occurance of a character from the end of a string**

Syntax:
```
strrtrim(string dest{}, int chr = ' ' )
```

Parameters:
dest: the string to pad. Required
chr: the character to remove. Optional. Defaults to the space character (0x20)

Usage:
```
new string a{} = "Hello World     ";
strrtrim(a);                        // variable a now contains the
string "Hello World"
new string b{} = "Hello World!!!!!";
strrtrim(b,"!");                    // variable b now contains
"Hello World"
```

Notes:
The single quote notation returns the numerical value of the ASCII character
between the single quotes.

strltrim

strltrim   removes any occurance of a character from the beginning of a string

Syntax:
```
strltrim(string dest{}, int chr = ' ' )
```

Parameters:
dest: the string to pad. Required
chr: the character to remove. Optional. Defaults to the space character (0x20)

Usage:
```
new string a{} = "     Hello World";
strltrim(a);                        // variable a now contains the
string "Hello World"
new string b{} = "!!!!!Hello World";
strltrim(b,"!");                    // variable b now contains
"Hello World"
```

Notes:
The single quote notation returns the numerical value of the ASCII character between the single quotes.

strtrim

strtrim removes any occurance of a character from the beginning and end of a string

Syntax:
```
strtrim(string dest{}, string chars{}=" ")
```

Parameters:
dest: the string to pad. Required
chars: the character to remove. Optional. Defaults to the space character (0x20)

Usage:
```
new string a{} = "    Hello World    ";
strtrim(a);                              // variable a now
contains the string "Hello World"
new string b{} = "!!!!!Hello World!!!!!";
strtrim(b,"!");                          // variable b now
contains "Hello World"
```

Notes:
The single quote notation returns the numerical value of the ASCII character between the single quotes.

strtolower

strtolower	**converts A-Z characters to a-z**

Syntax:
```
strtolower(string source{})
```

Parameters:
source: the string to convert

Usage:
```
new string a{} = "Hello World";
strtolower(a);                      // variable a now contains
"hello world"
```

Notes:
Only converts ASCII A-Z to a-z

strtoupper

strtoupper **converts a-z characters to A-Z**

Syntax:
```
strtoupper(string source{})
```

Parameters:
source: the string to convert

Usage:
```
new string a{} = "Hello World";
strtoupper(a);                    // variable a now contains "HELLO
WORLD"
```

Notes:
Only converts ASCII a-z to A-Z

Substring Methods

The following methods are useful when you only want to use a portion of the source string.

strmid

strmid   **copies a substring of a source string into a destination buffer**

Reserved. Not implemented yet.

strmidb

strmidb   **copies a substring of a source string into a destination buffer**

Syntax:
```
strmidb(dest{}, const source{}, start=0, end=cellmax,
maxsize=sizeof(dest))
```

Parameters:
dest: the buffer to store the substring. Required
source: the source string. Required
start: the zero-based **byte** index to start copying. Optional. Defaults to start of string.
end: the zero-based **byte** index to end copying. Optional. Defaults to end of string.
maxsize: the maximum number of 32-bit words to copy. Optional. Defaults to the maximum size of *dest* buffer

Usage:
```
new string a{} = "Hello World";
new string b{10};
strmidb(b,a);        // variable b now contains "Hello World"
strmidb(b,a,6);      // variable b now contains "World"
strmidb(b,a,0,5);    // variable b now contains "Hello"
```

Notes:
If operating on strings containing characters outside of the lower ASCII range, `strmidb` can actually start or end copying between two bytes of a multi-byte character, which corrupts the UTF-8 encoding. The function strmid is safer because operates on whole UTF-8 characters.

mid

mid	**returns a substring of the source string**

Reserved. Not implemented yet.

midb

midb	**returns a substring of the source string**

Syntax:
```
string midb(const source{}, start=0, end=cellmax)
```

Parameters:
source: the source string. Required
start: the zero-based byte index to start copying. Optional. Defaults to start of string.
end: the zero-based byte index to end copying. Optional. Defaults to end of string.

Returns:
A string array of size GEMscriptStringBufferSize containing the substring.

Usage:
```
new string a{} = "Hello World";
new string b{GEMscriptStringBuffer};
b = midb(a);            // variable b now contains "Hello World"
b = midb(a,6);          // variable b now contains "World"
b = midb(a,0,5);        // variable b now contains "Hello"
```

Notes:
If operating on strings containing characters outside of the lower ASCII range, midb can actually start or end copying between two bytes of a multi-byte character, which corrupts the UTF-8 encoding. The function mid is safer because operates on whole UTF-8 characters.

You may also use a "dot syntax" extension of the *source* string variable:

Syntax:
```
source.midb(start=0, end=cellmax)
```

Usage:
```
new string a{} = "Hello World"
new string b{GEMscriptStringBuffer}
b = a.midb(6) //variable b now contains "World"
b = a.midb(0,5) //variable b now contains "Hello"
```

strsplit

strsplit	**split string into two parts by some separator**

Syntax:
```
int strsplit(string dest{}, string source{}, int separator=' ',
int coalesce = true, maxsize=sizeof(dest))
```

Parameters:
dest: the destination buffer. Required
source: the source string. Required
separator: a 1-character string containing the delimiter character;
coalesce: if true, repeated separators are regarded as one.
maxsize: the maximum number of 32-bit words to copy. Optional. Defaults to the maximum size of *dest* buffer

Returns:
false (0) , if separator is not found or dest is too small. true (1) otherwise.

Usage:
```
new string a{} = "Hello World"
new string b{10}
//use default space
strsplit(b,a) // b = "Hello" and a = "World"

//use a new separator
new string c{} = "Hello World"
new string d{10}
strsplit(d,c,'o') //d = "Hell" c = " World"
//run it again with the same variables: d = "Hell" c = " World"
strsplit(d,c,'o') //d = " W" c = "rld"

//coalesce multiple separators in a row.
new string e{} = "Hello    World"
new string f{10}
strsplit(f,e,' ', true) //f = "Hello" and e = "World"
```

Notes:
The initial part is moved to dest and remaining part is left in the source. The separator is removed.
"true" is a macro which is equivalent to an integer type with value 1. "false" has a value of 0.

Search and Compare Methods

The following methods help with searching inside strings and comparing entire strings.

strcmp

strcmp
Syntax:
```
int strcmp(const string source1{}, const string source2{}, int
ignorecase=false, int length=cellmax)
```

Parameters:
source1: The first string to compare. Required.
source2: The second string to compare. Required.
ignorecase: determines if the two strings are deemed to be the same if they vary
only by case. Optional. Defaults to false.
length: The maximum number of bytes to compare. Optional. Defaults to longest
possible string.

Returns:
−1 if *source1* comes before *source2* (if *source1* is a substring of *source2*, then
return negative size difference)
1 if *source1* comes after *source2* (if *source2* is a substring of *source1*, then return
positive size difference), or
0 if the strings are equal (for the matched length).

Usage:
```
new string1{} = "Hello World";
new string2{} = "hello world";
new string3{} = "Hello";
new int a;
a = strcmp(string1,string2);         // a = -1 ('H'<'h')
a = strcmp(string1,string2,true);    // a = 0, without case,
strings are equal
a = strcmp(string1,string3, false, 5); // a = 0, first 5 bytes of
strings are equal
a = strcmp(string1,string3, false, 6); // a = 6 (string3 is a
substring of string1, so return with positive size difference)
a = strcmp(string3,string1, false);   // a = -6 (string3 is a
substring of string1, so return with negative size difference)
```

Notes:
ignorecase only applies to ASCII characters in the range 0x41-0x5A and
0x61-0x7A.

strfind

strfind

Reserved. Not implemented yet.

strfindb

strfindb
Syntax:
```
int strfind(const string source{}, const string sub{}, int
ignorecase=false, int index=0)
```

Parameters:
source: The string to search
sub: The substring to search for
ignorecase: determines if the match is deemed found if *sub* a portion of *source* vary only by case.
index: The byte offset within soure to start seaching. Optional. Defaults to the beginning of the string.

Returns:
The byte index of the first occurrence of the string *sub* in *source*, or −1 if no occurrence was found.
If an occurrence was found, you can search for the next occurrence by calling strfind again and set the parameter *index* to the returned value plus one.

Usage:
```
new string a{} = "Hello World";
new string b{} = "LL";
new int c;
c = strfindb(a,b);          //c = -1
c = strfindb(a,b,true);     //c = 2
c = strfindb(a,b,true,3); //c = -1
```

Notes:
ignorecase only applies to ASCII characters in the range 0x41-0x5A and 0x61-0x7A.

The index parameter is a byte offset, which is not the same as character offsets when using strings containing characters above the lower ASCII range. To use a UTF-8 character offset, use the function strfind

strequal

strequal

Syntax:
```
int strequal(const string source1{}, const string source2{}, int
ignorecase=false, int length=cellmax)
```

Parameters:
source1: The first string to compare. Required
source2: The second string to compare. Required
ignorecase: determines if the two strings are deemed to be the same if they vary only by case. Optioanl. Defaults to false.
length: The maximum number of bytes to compare. Optional. Defaults to the longest possible string.

Returns:
true, if the strings are equal. false if they are different.

Usage:
```
new string string1{} = "Hello World";
new string string2{} = "hello world";
new string string3{} = "Hello";
new int a;
a = strequal(string1,string2);          // a = false ('H'<'h')
a = strequal(string1,string2,true);     // a = true, without
case, strings are equal
a = strequal(string1,string3, false, 5); // a = true, first 5
bytes of strings are equal
a = strequal(string1,string3, false, 6); // a = false ( ' ' >
NULL)
```

Notes:
true is an int type with value 1, while false has a value of 0.

strchr

strchr   **Locate a byte in a string**

Reserved. Not implemented yet.

strchrb

strchrb   **Locate a byte in a string**

Syntax:
```
int strchr(const string source{}, int character, int start = 0)
```

Parameters:
source: The string to search. Required.
character: The character to search for. Required.
start: an offset to start searching. Optional. Defaults to the beginning of the string.

Returns:
the position as a 0-based index, or -1 if not found.

Usage:
```
new string source{} = "Hello World";
new int a;
a = strchrb(source, 'o');    // a = 4
a = strchrb(source, 'o', 5); // a = 7
```

Notes:
You can use the single-quoted formatting to specify the *character*: 'A' is the same as 0x41

If operating on strings containing characters outside of the lower ASCII range, strchrb can actually return an index which lies in between two bytes of a multi-byte character. Take care when using this index. The function strchr is safer because operates on whole UTF-8 characters.

strrchr

| **strrchr** | **Locate the last instance of a byte in a string.** |

Reserved. Not implemented yet.

strrchrb

strrchrb   **Locate the last instance of a byte in a string.**

Syntax:
```
int strrchrb(const string source{}, int character, int end =
cellmax)
```

Parameters:
source: The string to search. Required.
character: The character to search for. Required.
end: The byte offset to start searching from right to left. Optional. (Defaults to starting from end of string)

Returns:
the byte position as a 0-based index, or -1 if not found.

Usage:
```
new string source{} = "Hello World";
new int a;
a = strrchrb(source, 'o');    // a = 7
a = strrchrb(source, 'o', 6); // a = 4
```

Notes:
If operating on strings containing characters outside of the lower ASCII range, `strrchrb` can actually return an index which lies in between two bytes of a multi-byte character. Take care when using this index. The function strrchr is safer because operates on whole UTF-8 characters.

GEMscriptStringBufferSize

GEMscriptStringBufferSize
This is used by GEMscript API String methods which return a array. The GEMscript compiler can only allocate arrays of a static length at compile time, so any methods that returns a string via the stack (append, mid, itoa, and InternalRAM.string queries) will need to allocate this buffer at compile time. Each of these methods simply initializes a string buffer and calls the equivalent method that uses a destination buffer, then returns that buffer. For example, the implementation of itoa is:

```
string itoa(int number)
{
    new string dest{GEMscriptStringBufferSize} = "";
    valtostr(dest, number);
    return dest;
}
```

You can change the value of this macro in the Project -> Project Properties menu, under the GEMscript tab:

Error and warning messages

When the compiler finds an error it displays a "GEMscript Errors" dialog box with a list of single line error messages.

Each error message contains the following info:
• the name of the page containing the GEMscript error
• the line number where the compiler detected the error between parentheses, directly behind the page name
• the error class ("error", "fatal error" or "warning")
• an error number
• a descriptive error message

For example:
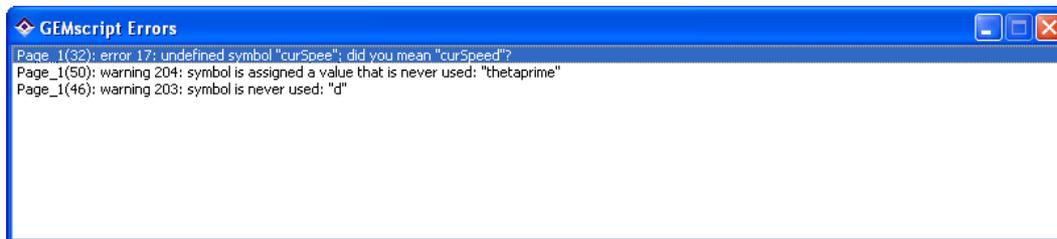
Note: the line number given by the compiler may specify a position behind the actual error, since the compiler cannot always establish an error before having analyzed the complete expression.

To view the source code that created the error, just double click on the error message. GEMstudio will then open the page that the error occured, launch the code functions editor for that page, and position the cursor to the line where the compiler detected the error. For example, clicking on the first error message in the GEMscript Errors dialog box will produce the following view of the page functions editor for Page_1:
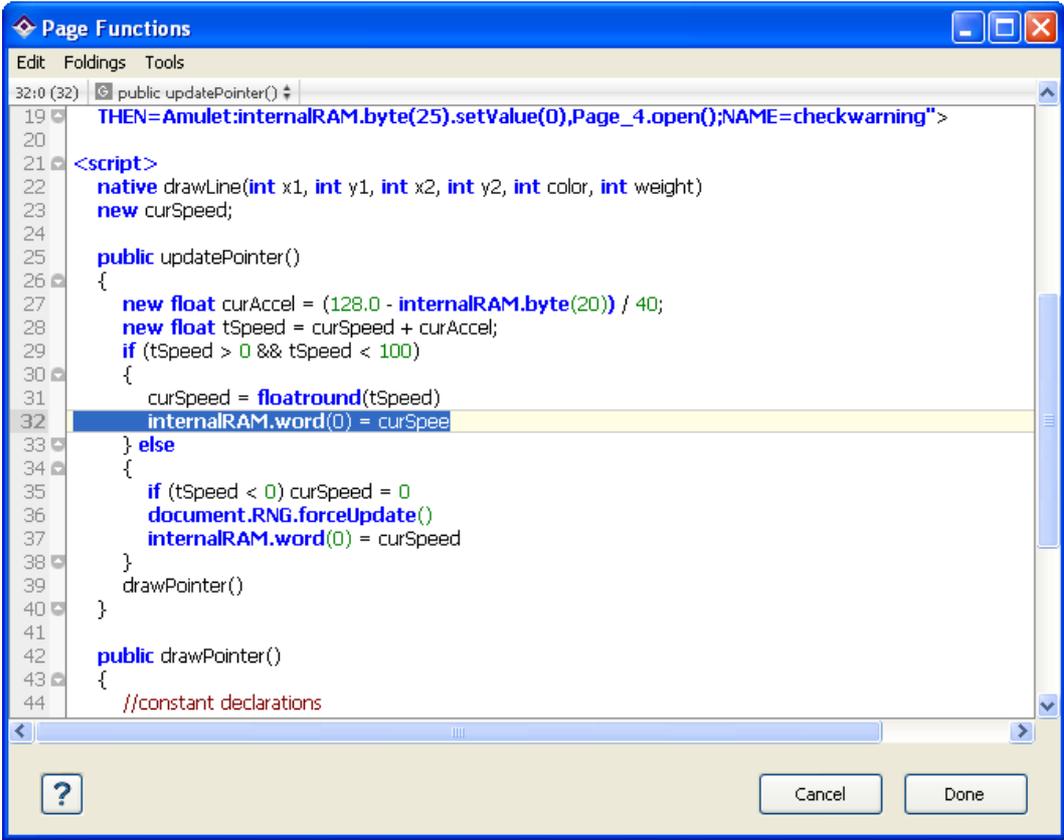
Page Functions

Edit Foldings Tools

32:0 (32) |G| public updatePointer()

```
19    THEN=Amulet:internalRAM.byte(25).setValue(0),Page_4.open();NAME=checkwarning">
20
21  <script>
22    native drawLine(int x1, int y1, int x2, int y2, int color, int weight)
23    new curSpeed;
24
25    public updatePointer()
26    {
27      new float curAccel = (128.0 - internalRAM.byte(20)) / 40;
28      new float tSpeed = curSpeed + curAccel;
29      if (tSpeed > 0 && tSpeed < 100)
30      {
31        curSpeed = floatround(tSpeed)
32        internalRAM.word(0) = curSpee
33      } else
34      {
35        if (tSpeed < 0) curSpeed = 0
36        document.RNG.forceUpdate()
37        internalRAM.word(0) = curSpeed
38      }
39      drawPointer()
40    }
41
42    public drawPointer()
43    {
44      //constant declarations
```

Cancel   Done

Error Categories

Errors are separated into three classes:

Errors
- Occurs in situations where the compiler is unable to generate appropriate code.
- Error messages are numbered from 1 to 99.

Fatal Errors
- Non recoverable error for which parsing must be aborted.
- Fatal error messages are numbered from 100 to 199.

Warnings
- Warnings are displayed for unintended compiler assumptions and common mistakes.
- Warning messages are numbered from 200 to 299.

Errors

001	expected token: *token*, but found *token* A required token is omitted.
002	only a single statement (or expression) can follow each "case" To put multiple statements in a case, you must enclose these statements between braces ("{. . .}") This creates a compound statement. Without the braces, each case in a switch statement can only hold one statement.
003	declaration of a local variable must appear in a compound block The declaration of a local variable must appear between braces("{. . .}") at the active scope level. When the parser flags this error, a variable declaration appears as the only statement of a function or the only statement below an `if`, `else`, `for`, `while` or `do` statement. Note that, since local variables are accessible only from (or below) the scope that their declaration appears in, having a variable declaration as the only statement at any scope is useless.
004	function name is not implemented. There is no implementation for the designated function. The function may have been "forwardly" declared, or prototyped, but the full function definition including a statement or statement block is missing.
005	function may not have arguments Arguments are not allowed on this function.
006	must be assigned to an array String literals or arrays must be assigned to an array. This error message may also indicate a missing index (or indices) at the array on the right side of the "=" sign.
007	operator cannot be redefined Only a select set of operators may be redefined, this operator is not one of them.
008	must be a constant expression; assumed zero The size of arrays and the parameters of most directives must be constant values.

009	invalid array size (negative, zero or out of bounds)
	The number of elements of an array must always be 1 or more. In addition, an array so big that it exceeds the 32-bit number range is invalid too.
010	illegal function or declaration
	The compiler expects a declaration of a global variable or of a function at the current location, but it cannot interpret it as such.
011	invalid outside functions
	The instruction or statement is invalid at a global level. Local labels and (compound) statements are only valid if used within functions.
012	invalid function call, not a valid address
	The symbol is not a function.
013	no entry point (no public functions)
	The file does not contain a public function. The compiled file thereby does not have any entry points for execution.
014	invalid statement; not in switch
	The statements `case` and `default` are only valid inside a `switch` statement.
015	"`default`" must be the last clause in switch statement
	GEMscript requires the `default` clause to be the last clause in a `switch` statement.
016	multiple defaults in "switch"
	Each `switch` statement may only have one default clause.
017	undefined symbol *symbol*
	The symbol (variable, constant or function) is not declared.
018	initialization data exceeds declared size
	An array with an explicit size is initialized, but the number of initializers exceeds the number of elements specified. For example,

	in "`arr[3]={1,2,3,4};`" the array is specified to have three elements, but there are four initializers.
019	not a label: *name*
	A goto statement branches to a symbol that is not a label.
020	invalid symbol *name*
	A symbol may start with a letter, an underscore or an "at" sign ("@") and may be followed by a series of letters, digits, underscore characters and "@" characters.
021	symbol already defined: *identifier*
	The symbol was already defined at the current level.
022	must be lvalue (non-constant)
	The symbol that is altered (incremented, decremented, assigned a value, etc.) must be a variable that can be modified (this kind of variable is called an lvalue). Functions, string literals, arrays and constants are not lvalues. Variables declared with the "const" attribute are no lvalues either.
023	array assignment must be simple assignment
	When assigning one array to another, you cannot combine an arithmetic operation with the assignment (e.g., you cannot use the "+=" operator).
024	"break" or "continue" is out of context
	The statements `break` and `continue` are only valid inside the context of a loop (a `do`, `for` or `while` statement). Unlike the languages C/C++ and Java, break does not jump out of a switch statement.
025	function heading differs from prototype
	The number of arguments given at a previous declaration of the function does not match the number of arguments given at the current declaration.
026	no matching "#if..."
	The directive `#else` or `#endif` was encountered, but no matching `#if` directive was found.
027	invalid character constant

	One likely cause for this error is the occurrence of an unknown escape sequence, like "\x". Putting multiple characters between single quotes, as in 'abc' also issues this error message. A third cause for this error is a situation where a character constant was expected, but none (or a non-character expression) were provided.
028	invalid subscript (not an array or too many subscripts): *identifier* The subscript operators "[" and "]" are only valid with arrays. The number of square bracket pairs may not exceed the number of dimensions of the array.
029	invalid expression, assumed zero The compiler could not interpret the expression.
030	compound statement not closed at the end of file (started at line number) An unexpected end of file occurred. One or more compound statements are still unfinished (i.e. the closing brace "}" has not been found). The line number where the compound statement started is given in the message.
031	unknown directive The character "#" appears first at a line, but no valid directive was specified.
032	array index out of bounds The array index is larger than the highest valid entry of the array.
033	array must be indexed (*variable name*) An array as a whole cannot be used in a expression; you must indicate an element of the array between square brackets.
034	argument does not have a default value (*argument index*) You can only use the argument placeholder when the function definition specifies a default value for the argument.
035	argument type mismatch (*argument index*) The argument that you pass is different from the argument that the function expects, and the compiler cannot convert the passed-in argument to the required type. For example, you cannot pass the

literal value "1" as an argument when the function expects an array or a reference.

036	empty statement
	The line contains a semicolon that is not preceded by an expression. GEMscript does not support a semicolon as an empty statement, use an empty compound block instead.
037	invalid string (possibly non-terminated string)
	A string was not well-formed; for example, the final quote that ends a string is missing.
038	extra characters on line
	There were trailing characters on a line that contained a directive (a directive starts with a # symbol).
039	constant symbol has no size
	A variable has a size, a constant has no size. That is, you cannot use a (symbolic) constant with the `sizeof` operator, for example.
040	duplicate "case" label (value *value*)
	A preceding "case label" in the list of the switch statement evaluates to the same value.
041	invalid ellipsis, array size is not known
	You used a syntax like "`arr[] = { 1, ... };`", which is invalid because the compiler cannot deduce the size of the array from the declaration.
042	invalid combination of class specifiers
	A function or variable is denoted as both "`public`" and "`native`", which is unsupported. Other combinations may also be unsupported; for example, a function cannot be both "`public`" and "`stock`" (a variable may be declared both "`public`" and "`stock`").
043	character constant value exceeds range for a packed string/array
	When the error occurs on a literal array, it is usually an attempt to store a value larger than 8-bits in an element of a packed array, where each element is 8-bits.
044	positional parameters must precede all named parameters

	When you mix positional parameters and named parameters in a function call, the positional parameters must come first.
045	too many function arguments
	The maximum number of function arguments is currently limited to 64.
046	unknown array size (variable *name*)
	For array assignment, the size of both arrays must be explicitly defined, also if they are passed as function arguments.
047	array sizes do not match, or destination array is too small
	For array assignment, the arrays on the left and the right side of the assignment operator must have the same number of dimensions. In addition: • for multi-dimensional arrays, both arrays must have the same size —note that an unpacked array does not fit in a packed array with the same number of elements; • for single arrays with a single dimension, the array on the left side of the assignment operator must have a size that is equal or bigger than the one on the right side. When passing arrays to a function argument, these rules also hold for the array that is passed to the function (in the function call) versus the array declared in the function definition. When a function returns an array, all return statements must specify an array with the same size and dimensions.
048	array dimensions do not match
	For an array assignment, the dimensions of the arrays on both sides of the "=" sign must match; when passing arrays to a function argument, the arrays passed to the function (in the function call) must match with the definition of the function arguments. When a function returns an array, all return statements must specify an array with the same size and dimensions.
049	invalid line continuation
	A line continuation character (a backslash at the end of a line) is at an invalid position, for example at the end of a file or in a single line comment.
050	invalid range

	A numeric range with the syntax "n1 .. n2", where n1 and n2 are numeric constants, is invalid. Either one of the values in not a valid number, or n1 is not smaller than n2.
051	invalid subscript, use "[ ]" operators on major dimensions and for named indices You can use the "character array index" operator (braces: "{ }" only for the last dimension, and only when indexing the array with a number. For other dimensions, and when indexing the array with a "symbolic index" (one that starts with a "."), you must use the cell index operator (square brackets: "[ ]").
052	multi-dimensional arrays must be fully initialized If an array with more than one dimension is initialized at its declaration, then there must be equally many literal vectors/subarrays at the right of the equal sign ("=") as specified for the major dimension(s) of the array.
053	exceeding maximum number of dimensions The current implementation of the GEMscript compiler only supports arrays with one or two dimensions.
054	unmatched closing brace A closing brace ("}") was found without matching opening brace ("{").
055	start of function body without function header An opening brace ("{") was found outside the scope of a function. This may be caused by a semicolon at the end of a preceding function header.
056	arrays, local variables and function arguments cannot be public A local variable or a function argument starts with the character "@", which is invalid.
057	Unfinished expression before compiler directive Compiler directives may only occur between statements, not inside a statement. This error typically occurs when an expression statement is split over multiple lines and a compiler directive appears between the start and the end of the expression. This is not supported.
058	duplicate argument; same argument is passed twice

	In the function call, the same argument appears twice, possibly through a mixture of named and positional parameters.
059	function argument may not have a default value (variable *name*) All arguments of public functions must be passed explicitly. Public functions are called from the Amulet OS, which has no knowledge of the default parameter values. Arguments of user defined operators are implied from the expression and cannot be inferred from the default value of an argument.
060	multiple "#else" directives between "#if . . . #endif" Two or more `#else` directives appear in the body between the matching `#if` and `#endif`.
061	"#elseif" directive follows an "#else" directive All `#elseif` directives must appear before the `#else` directive. This error may also indicate that an `#endif` directive for a higher level is missing.
062	number of operands does not fit the operator When redefining an operator, the number of operands that the operator has (1 for unary operators and 2 for binary operators) must be equal to the number of arguments of the operator function.
063	function result type of operator *name* must be *name* Logical and relational operators are defined as having a result that is either true (1) or false (0) and having a "bool:" type. A user defined operator should adhere to this definition.
064	cannot change predefined operators One cannot define operators to work on untagged values, for example, because GEMscript already defines this operation.
065	function argument may only have a single tag (*argument number*) In a user defined operator, a function argument may not have multiple types.
066	function argument may not be a reference argument or an array (*argument number*) In a user defined operator, all arguments must be non-array types that are passed "by value".

067	variable cannot be both a reference and an array (*variable name*)
	A function argument may be denoted as a "reference" or as an array, but not as both.
068	invalid rational number precision in #pragma
	For floating point rational numbers, the precision specification should be omitted.
069	rational number format already defined
	This #pragma conflicts with an earlier #pragma that specified a different format.
070	rational number support was not enabled
	A rational literal number was encountered, but the format for rational numbers was not specified.
071	user-defined operator must be declared before use (*function name*)
	Like a variable, a user-defined operator must be declared before its first use. This message indicates that prior to the declaration of the user-defined operator, an instance where the operator was used on operands with the same types occurred. This may either indicate that the program tries to make mixed use of the default operator and a user-defined operator (which is unsupported), or that the user-defined operator must be "forwardly declared".
072	"sizeof" operator is invalid on "function" symbols
	You used something like "`sizeof MyCounter`" where the symbol "`MyCounter`" is not a variable, but a function. You cannot request the size of a function.
073	function argument must be an array (*argument name*)
	The function argument is a constant or a simple variable, but the function requires that you pass an array.
074	#define pattern must start with an alphabetic character
	Any pattern for the `#define` directive must start with a letter, an underscore ("_") or an "@"-character. The pattern is the first word that follows the `#define` keyword.
075	input line too long (after substitutions)

Either the source file contains a very long line, or text substitutions make a line that was initially of acceptable length grow beyond its bounds. This may be caused by a text substitution that causes recursive substitution (the pattern matching a portion of the replacement text, so that this part of the replacement text is also matched and replaced, and so forth).

076	syntax error in the expression, or invalid function call The expression statement was not recognized as a valid statement (so it is a "syntax error"). From the part of the string that was parsed, it looks as if the source line contains a function call in a "procedure call" syntax (omitting the parentheses), but the function result is used — assigned to a variable, passed as a parameter, used in an expression. . .
077	malformed UTF-8 encoding, or corrupted file: filename The file starts with an UTF-8 signature, but it contains encodings that are invalid UTF-8. If the source file was created by an editor or converter that supports UTF-8, the UTF-8 support is non-conforming.
078	function uses both "return" and "return <value>" The function returns both *with and without* a return value. The function should be consistent in always returning with a function result, or in never returning a function result.
079	inconsistent return types (array & non-array) The function returns both values and arrays, which is not allowed. If a function returns an array, all `return` statements must specify an array (of the same size and dimensions).
080	unknown symbol, or not a constant symbol (symbol *name*) Where a constant value was expected, an unknown symbol or a non-constant symbol (variable) was found.
082	user-defined operators and native functions may not have states Only standard and public functions may have states.
083	a function or variable may only belong to a single automaton (symbol *name*) There are multiple automatons in the state declaration for the indicated function or variable, which is not supported. In the case of a function: all instances of the function must belong to the same automaton. In the case of a variable: it is allowed to have several

	variables with the same name belonging to different automatons, but only in separate declarations — these are distinct variables.
084	state conflict: one of the states is already assigned to another implementation (symbol *name*) The specified state appears in the state specifier of two implementations of the same function.
085	no states are defined for symbol *name* When this error occurs on a function, this function has a fall-back implementation, but no other states. If the error refers to a variable, this variable does not have a list of states between the < and > characters. Use a state-less function or variable instead.
086	unknown automaton *name* The "`state`" statement refers to an unknown automaton.
087	unknown state name for automaton *name* The "`state`" statement refers to an unknown state (for the specified automaton).
088	public variables and local variables may not have states (symbol *name*) Only standard (global) variables may have a list of states (and an automaton) at the end of a declaration.
089	state variables may not be initialized (symbol *name*) Variables with a state list may not have initializers. State variables should always be initialized through an assignment (instead of at their declaration), because their initial value is indeterminate.
090	public functions may not return arrays (symbol *name*) A public function may not return an array. Returning arrays is allowed only for normal functions.
091	first constant in an enumerated list must be initialized (symbol *name*) The first constant in a list of enumerated symbolic constants must be set to a value. Any subsequent symbol is automatically set the the value of the preceding symbol +1.
092	invalid number format A symbol started with a digit, but is is not a valid number.

| 093 | array fields with a size may only appear in the final dimension

In the final dimension (the "minor" dimension), the fields of an array may optionally be declared with a size that is not 32-bit. On the major dimensions of an array, this is not valid, however. |
|-----|----|
| 094 | invalid subscript, subscript does not match array definition regarding named indices (symbol name)

Either the array was declared with symbolic subscripts and you are indexing it with an expression, or you are indexing the array with a symbolic subscript which is not defined for the array. |

Fatal Errors

100	cannot read from file: *filename* The compiler cannot find the specified file or does not have access to it.
101	cannot write to file: *filename* The compiler cannot write to the specified output file, probably caused by insufficient disk space or restricted access rights (the file could be read-only, for example).
102	table overflow: *table name* An internal table in the GEMscript parser is too small to hold the required data. Some tables grow dynamically, which means that there was insufficient memory to resize the table. The "table name" is one of the following: • **staging buffer:** The staging buffer holds the code generated for an expression before it is passed to the peephole optimizer. Because the staging buffer grows dynamically, an overflow of the staging buffer is basically an "out of memory" error. • l**oop table:** The loop table is a stack used with nested do, for, and while statements. The table allows nesting of these statements up to 24 levels. • **literal table:** This table keeps the literal constants (numbers, strings) that are used in expressions and as initializers for arrays. The literal table grows dynamically, so an overflow of the literal table basically is an "out of memory" error. • **compiler stack:** The compiler uses a stack to store temporary information it needs while parsing. An overflow of this stack is probably caused by deeply nested (or recursive) file inclusion. The compiler stack grows dynamically, so an overflow of the compiler stack basically is an "out of memory" error. • **option table:** In case that there are more options on the command line or in the response file than the compiler can cope with.
103	insufficient memory General "out of memory" error.
104	incompatible options: *option* versus *option* Two option that are passed to the GEMscript compiler conflict with each other, or an option conflicts with the configuration of the GEMscript compiler.
105	numeric overflow, exceeding capacity

	A numeric constant, notably a dimension of an array, is too large for the compiler to handle.
106	compiled script exceeds the maximum memory size (number bytes) The memory size for the abstract machine that is needed to run the script exceeds the value set with `#pragma amxlimit`. This means that the script is too large to be supported by the Amulet OS. This can be remedied by removing DRAM utilization in the GUI side, but if that is not possible then there are strategies to reduce the size of your code. You might try reducing the script's memory requirements by: • setting a smaller stack/heap area —see `#pragma dynamic`; • putting repeated code in separate functions; • putting repeated data (strings) in global variables; • trying to find more compact algorithms to perform the same task.
107	too many error/warning messages on one line A single line that causes several error/warning messages is often an indication that the GEMscript parser is unable to "recover" from an earlier error. In this situation, the parser is unlikely to make any sense of the source code that follows —producing only (more) inappropriate error messages. Therefore, compilation is halted.
108	codepage mapping file not found You may not change the codepage from UTF-8
109	invalid path: *path name* A path, for example for include files, is invalid. Check the compiler options and, if used, the configuration file.
110	assertion failed: *expression* Compile-time assertion failed.
111	user error: message The parser fell on an `#error` directive.
112	overlay function *name* exceeds limit by *value* bytes Overlays are not currently supported by GEMscript

Warnings

200	symbol is truncated to *number* characters
	The symbol is longer than the maximum symbol length. The maximum length of a symbol depends on whether the symbol is native, public or neither. Truncation may cause different symbol names to become equal, which may cause error 021 or warning 219.
201	redefinition of constant/macro (symbol *name*)
	The symbol was previously defined to a different value, or the text substitution macro that starts with the prefix name was redefined with a different substitution text.
202	number of arguments does not match definition
	At a function call, the number of arguments passed to the function (actual arguments) differs from the number of formal arguments declared in the function heading. To declare functions with variable argument lists, use an ellipsis (...) behind the last known argument in the function heading; for example: `int sum(int a,...);`
203	symbol is never used: identifier
	A symbol is defined but never used. Public functions are excluded from the symbol usage check (since these may be called from the outside).
204	symbol is assigned a value that is never used: *identifier*
	A value is assigned to a symbol, but the contents of the symbol are never accessed.
205	redundant code: constant expression is zero
	Where a conditional expression was expected, a constant expression with the value zero was found, e.g. "`while (0)`" or "`if (0)`". The conditional code below the test is never executed, and it is therefore redundant.
206	redundant test: constant expression is non-zero
	Where a conditional expression was expected, a constant expression with a non-zero value was found, e.g. `if (1)`. The test is redundant, because the conditional code is always executed. To create an endless loop, use `for ( ;; )` instead of `while (1)`.
207	unknown "#pragma"

The compiler ignores the pragma. The `#pragma` directives may change between compilers of different vendors and between different versions of a compiler of the same version.

208	function with type result used before definition, forcing reparse When a function is "used" (invoked) before being declared, the return type is assumed integer. If that function returns a value with a different type, the parser must make an extra pass over the source code because the presence of the type may change the interpretation of operators such as floating point or integer (or in the presence of user-defined operators). You can speed up the parsing/compilation process by declaring the relevant functions before using them.
209	function should return a value The function does not have a return statement, or it does not have an expression behind the return statement, but the function's result is used in a expression.
210	possible use of symbol before initialization: *identifier* A local uninitialized variable appears to be read before a value is assigned to it. The compiler cannot determine the actual order of reading from and storing into variables and bases its assumption of the execution order on the physical appearance order of statements an expressions in the source file.
211	possibly unintended assignment A conditional expression was expected, but the assignment operator `(=)` was found instead of the equality operator `(==)`. Since this is a frequent mistake, the compiler issues a warning. To avoid this message, put parentheses around the expression, e.g. `if ( (a=2) )`.
212	possibly unintended bitwise operation A conditional expression was expected, but a bitwise operator `(&` `or |)` was found instead of a Boolean operator `(&& or ||)`. In situations where a bitwise operation seems unlikely, the compiler issues this warning. To avoid this message, put parentheses around the expression.
214	possibly a "const" array argument was intended: *identifier* Arrays are always passed by reference. If a function does not modify the array argument, however, the compiler can sometimes generate more compact and quicker code if the array argument is specifically marked as "const".

215	expression has no effect The result of the expression is apparently not stored in a variable or used in a test. The expression or expression statement is therefore redundant.
216	nested comment GEMscript may not always properly handle nested comments.
217	loose indentation Statements at the same logical level do not start in the same column; that is, the indents of the statements are different. Although GEMscript is a free format language, loose indentation frequently hides a logical error in the control flow. You can also disable this warning with `#pragma tabsize 0` or the compiler option -t:0.
218	old style prototypes used with optional semicolon When using "optional semicolons", it is preferred to explicitly declare forward functions with the forward keyword than using terminating semicolon.
219	local variable identifier shadows a symbol at a preceding level A local variable has the same name as a global variable, a function, a function argument, or a local variable at a lower precedence level. This is called "shadowing", as the new local variable makes the previously defined function or variable inaccessible. Note: if there are also error messages further on in the script about missing variables (with these same names) or brace level problems, it could well be that the shadowing warnings are due to these syntactical and semantical errors. Fix the errors first before looking at the shadowing warnings.
220	expression with tag override must appear between parentheses In a case statement and in expressions in the conditional operator (`" ? : "`), any expression that has a tag override should be enclosed between parentheses, to avoid the colon to be misinterpreted as a separator of the case statement or as part of the conditional operator.
221	label name *identifier* shadows tag name

	A code label (for the `goto` instruction) has the same name as a previously defined tag. This may indicate a improperly applied tag override; a typical case is an attempt to apply a tag override on the variable on the left of the = operator in an assignment statement.
222	number of digits exceeds rational number precision

A literal rational number has more decimals in its fractional part than the precision of a rational number supports. The remaining decimals are ignored. |
| 223 | redundant "sizeof": argument size is always 1 (symbol *name*)

A function argument has as its default value the size of another argument of the same function. The "`sizeof`" default value is only useful when the size of the referred argument is unspecified in the declaration of the function; i.e., if the referred argument is an array. |
| 224 | indeterminate array size in "sizeof" expression (symbol *name*)

The operand of the `sizeof` operator is an array with an unspecified size. That is, the size of the variable cannot be determined at compile time. If used in an "`if`" instruction, consider a conditionally compiled section, replacing if by `#if`. |
| 225 | unreachable code

The indicated code will never run, because an instruction before (above) it causes a jump out of the function, out of a loop or elsewhere. Look for `return, break, continue` and `goto` instructions above the indicated line. Unreachable code can also be caused by an endless loop above the indicated line. |
| 226 | a variable is assigned to itself (symbol *name*)

There is a statement like "`x = x`" in the code. The parser checks for self assignments after performing any text and constant substitutions, so the left and right sides of an assignment may appear to be different at first sight. For example, if the symbol "`TWO`" is a constant with the value 2, then "`var[TWO] = var[2]`" is also a self-assignment.

Self-assignments are, of course, redundant and they may hide an error (assignment to the wrong variable, error in declaring constants).

Note that the GEMscript parser is limited to performing "static checks" only. In this case it means that it can only compare array assignments for self-assignment with constant array indices. |
| 227 | more initializers than array fields |

	An array that is declared with symbolic subscripts contains more values/fields as initializers than there are (symbolic) subscripts.
228	length of initializer exceeds size of the array field The initializer for an array element contains more values than the size of that field allows. This occurs in an array that has symbolic subscripts, and where a particular subscript is declared with a size.
229	mixing packed and unpacked array indexing or array assignment An array is declared as packed (with { and } braces) but indexed as unpacked (with [and]), or vice versa. Or one array is assigned to another and one is packed while the other is unpacked.
230	no implementation for state name in function name, no fall-back A function is lacking an implementation for the indicated state. The compiler cannot statically check whether the function will ever be called in that state, and therefore it issues this warning. When the function would be called for the state for which no implementation exists, the abstract machine aborts with a run time error.
231	state specification on forward declaration is ignored A state specification is redundant on forward declarations. The function signature must be equal for all states. Only the implementations of the function are state-specific.
232	native function lacks a predefined index (symbol I) The GEMscript compiler was configured with predefined indices for native functions, but it encountered a declaration for which it does not have an index declaration. This usually means that the script uses include files that are not appropriate for the active configuration.
233	state variable name shadows a global variable The state variable has the same name as a global variable (without state specifiers). This means that the global variable is inaccessible for a function with one of the same states as those of the variable.
234	function is deprecated (symbol *name*) The script uses a function which as marked as "deprecated". GEMstudio can mark API functions as deprecated when better alternatives for the function are available or if the function may not be supported in future versions of the Amulet OS.
235	public function lacks forward declaration (symbol *name*)

	The script defines a public function, but no forward declaration of this function is present. Possibly the function name was written incorrectly. The requirement for forward declarations of public functions guards against a common error.
236	unknown parameter in substitution (incorrect #define pattern) A `#define` pattern contains a parameter in the replacement (e.g. "`%1`"), that is not in the match pattern. See the preprocessor section for proper syntax.
237	recursive function *name* The specified function calls itself recursively. Although this is valid in GEMscript, a self-call is often an error. Note that this warning is only generated when the GEMscript parser/compiler is set to "verbose" mode.
238	mixing string formats in concatenation In concatenating literals strings, strings with different formats (such as packed versus unpacked, and "plain" versus standard strings) were combined. This is usually an error. The parser uses the format of the first (left-most) string in the concatenation for the result.