



# Reference

Version 1.17

m Mobile Shell, Reference, Version 1.17  
Written by Lukas Knecht

[www.m-shell.net](http://www.m-shell.net)

Document IW-M-REF-1.31

© 2004-2007 infowing AG, 8703 Erlenbach, Switzerland

The information contained herein is the property of infowing AG and shall neither be reproduced in whole or in part without prior written approval from infowing AG. All rights are reserved, whether the whole or part of the material is concerned, specifically those of translation, reprinting, reuse of illustration, broadcasting, reproduction by photocopying machine or similar means and storage in data banks. infowing AG reserves the right to make changes, without notice, to the contents contained herein and shall not be responsible for any damages (including consequential) caused by reliance on the material as presented.

Typeset in Switzerland.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Language</b>	<b>5</b>
2.1	Data Types . . . . .	5
2.2	Comments . . . . .	6
2.3	Literals . . . . .	7
2.4	Variables . . . . .	10
2.5	Arrays . . . . .	11
2.6	Expressions . . . . .	13
2.7	Statements . . . . .	18
2.7.1	Assignment . . . . .	19
2.7.2	Increment . . . . .	20
2.7.3	If Statement . . . . .	21
2.7.4	While Statement . . . . .	22
2.7.5	Do-Until Statement . . . . .	23
2.7.6	For Statement . . . . .	23
2.7.7	Case Statement . . . . .	25
2.7.8	Break Statement . . . . .	27
2.7.9	Return Statement . . . . .	27
2.7.10	print Statement . . . . .	27
2.8	Functions . . . . .	29
2.9	Modules . . . . .	34
2.10	Exceptions . . . . .	38
2.11	Source Structure . . . . .	40

<b>3</b>	<b>Library</b>	<b>41</b>
3.1	Path and File Names . . . . .	41
3.2	Builtin Functions and Constants . . . . .	43
3.3	Module <code>array</code> : Array Functions . . . . .	55
3.4	Module <code>audio</code> : Audio Functions . . . . .	63
3.5	Module <code>contacts</code> : Contacts Database . . . . .	70
3.6	Module <code>files</code> : File and Directory Access . . . . .	78
3.7	Module <code>graph</code> : Screen Graphics . . . . .	87
3.8	Module <code>gsm</code> : GSM information . . . . .	108
3.9	Module <code>io</code> : File and Stream Input/Output . . . . .	111
3.10	Module <code>math</code> : Mathematical Functions . . . . .	121
3.11	Module <code>sms</code> : Short Messages . . . . .	125
3.12	Module <code>system</code> : System Related Functions . . . . .	129
3.13	Module <code>time</code> : Time and Date Functions . . . . .	131
3.14	Module <code>ui</code> : User Interface Functions . . . . .	135
<b>4</b>	<b>Interactive Shells</b>	<b>149</b>
4.1	Simplified Syntax for Interactive Use . . . . .	149
4.2	Shell Builtin Functions . . . . .	150
<b>5</b>	<b>SMS Control</b>	<b>155</b>
<b>A</b>	<b>Appendix</b>	<b>157</b>
A.1	Exception Tags . . . . .	157
A.2	Reserved words . . . . .	161
A.3	Properties (.prp) File . . . . .	161
A.4	User Permissions . . . . .	165
	<b>Index</b>	<b>167</b>

# 1. Introduction

m is a simple and easy to learn programming language intended for mobile phones (“Smart Phones”). m has been specifically designed for the limited text editing capabilities of these devices. The language thus has few special characters, and the library functions generally use short identifiers.

To obtain a flat learning curve, in particular for the novice user, and to keep editing m code manageable on a cell phone, the m language has been kept simple, while still providing a rich set of programming constructs and functions.

Likewise, the library of modules closely reflects the capabilities of smart phones. Modules have been designed with ease of use in mind, without requiring complex setup operations or even an understanding of the underlying architecture.

To protect the phone’s data, the user’s purse, and the phone’s integrity from malevolent scripts, permissions to use potentially dangerous functions are configurable.



## 2. Language

This chapter defines the `m` programming language. `m` is a procedural language supporting code reuse through a simple concept of modules.

The following sections introduce the building blocks of `m`. After each section, the `m` syntax is summarized by a formal definition in EBNF (Extended Backus Naur Form):

- Text in single quotes `' '` corresponds to the actual text (terminal symbols).
- Text in bold face denotes keywords (reserved words).
- The vertical bar `|` separates alternatives.
- Text in brackets `[]` is optional.
- Text in curly braces `{ }` can be repeated (zero times, once or many times).
- Text in parentheses `( )` is grouped together.

`m` scripts are read as a series of tokens which are separated by “separator” characters (all characters which are not letters, digits or an underscore). White space (blank and new line) always separates two characters. The amount of white space used does not affect the meaning of a script, but white space should be added sensibly to make a script more readable by indenting lines to reflect the structure of the code.

### 2.1 Data Types

`m` supports the following data types:

- **Number:** numbers have a range of roughly  $-10^{308}$  to  $10^{308}$  and have a precision of almost 17 decimal digits<sup>1</sup>.
- **String:** strings are sequences of characters<sup>2</sup>. Strings are immutable: their length is fixed, and individual characters cannot be changed. However, there are many builtin functions (see [builtin functions](#) (p. 43)) manipulating strings.
- **Boolean:** booleans are logical values, i.e. either `true` or `false`. For instance, the result of a comparison is of Boolean type. Booleans are also often used as flags or to denote options for functions.
- **Array:** arrays are collections of arbitrarily many values. Multidimensional arrays (e.g. matrices) are constructed as arrays of arrays. In `m`, arrays are dynamic in size. Elements can be appended or removed. Elements can be indexed by numbers or strings (“associative array”). See also section 2.5 (p. 11).
- **Function Reference:** a reference (“pointer”) to a function. The reference can be used to specify callback functions, or to implement a simple polymorphism scheme.
- **Null:** this special type denotes an uninitialized or unspecified value. The only value of this type is `null`.
- **Native Objects:** are created by modules which are tied closely into the underlying operating system, e.g. by module `io` (p. 111). Native objects can only be assigned and compared for identity.

## 2.2 Comments

Normally, all characters in an `m` script are assumed to be `m` language. Comments intended for the human reader must therefore be specially marked:

- Single line comments start with a double slash: any text from a `//` to the end of the line containing it is considered a comment.

---

<sup>1</sup>Internally, numbers are stored as 64 bit floating point values in IEEE format, with 52+1 bit mantissa and 11 bit exponent.

<sup>2</sup>Internally, each character is represented by 16 bits, thus supporting the UNICODE<sup>®</sup> basic multilingual plane. However, fonts often support only the ISO-8859-1 (Latin) character set.



- Multiline comments start with slash-star and end with star-slash: any text between `/*` and `*/` is considered a comment and ignored. These comments can be nested.

```
print 3*3 // this prints nine
→ 9
/* The following m code is within this comment,
   so it is ignored:
print 5/7
   This is still part of the comment.
   /* This is a nested comment ending here: */
   This is the last line of this comment. */
print 3/4
→ 0.75
```

Comment marks cannot be placed within string literals (see section 2.3 (p. 7)).

## 2.3 Literals

Literals are concrete values specified explicitly in the code. Except for array literals, they are fixed and cannot change during script execution. Array literals are more complex and discussed in section 2.5 (p. 11).

```
| SimpleLiteral := NumberLiteral | StringLiteral | BooleanLiteral |
| FunctionLiteral | NullLiteral .
```

### Number Literals

A number literal is a sequence of digits, with an optional decimal point, and an optional decimal exponent. The digits must not be separated by white space or thousands separators:

```

print 0
→ 0
print 3.1415927
→ 3.1415927
print 6.02214199e+23
→ 6.022142E+23
print 1E-3
→ 0.001

```

Integer numbers can also be written in hexadecimal notation, by prefixing them with 0x:

```

print 0xff
→ 255
print 0x1000
→ 4096

```

```

NumberLiteral :=
  Digit {Digit} ['.' {Digit}]
  [(E' | 'e') ['- ' | '+'] Digit {Digit}] |
  '0x' HexDigit {HexDigit} .
Digit :=
  '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
HexDigit := Digit | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
  'a' | 'b' | 'c' | 'd' | 'e' | 'f' .

```

## String Literals

A string literal is a sequence of characters between single or double quotes.

```

print 'Hello, world!'
→ Hello, world!
print "That's nice"
→ That's nice

```

In order to produce all characters, the backslash \ serves as escape for the following character. For instance, if the quote used to delimit the string literal occurs inside the string, it must be escaped. Likewise, the backslash itself must be escaped, as is often seen in path names:

```
print "A quote: \"To be, or not to be...\""
→ A quote: "To be, or not to be..."
print 'That\'s nice'
→ That's nice
print "c:\\system\\apps"
→ c:\system\apps
```

There are a few characters which have a special meaning when escaped:

<code>\f</code>	form feed (ASCII 12)
<code>\n</code>	new line or line feed (ASCII 10)
<code>\r</code>	carriage return (ASCII 13)
<code>\t</code>	horizontal tab (ASCII 9)
<code>\u</code>	hexadecimal UNICODE <sup>®</sup> follows

```
print "Line1\nLine2"
→ Line1
   Line2
print "Item1\tItem2"
→ Item1   Item2
print "g\u00e9nial"
→ génial
```

The maximum length of a string literal is 256 characters.

```
StringLiteral := '"' {Char | EscapeChar | "\"} '"' |
               "'" {Char | EscapeChar | '\''} "'" .
Char := (printable ISO-8859-1 char except ' , " , \)
EscapeChar := '\ ' ('n' | 'r' | 't' |
                  'u' HexDigit HexDigit HexDigit HexDigit | (printable char)) .
```

## Boolean Literals

Not surprisingly, there are just two boolean literals: `true` and `false`.

```
BooleanLiteral := false | true .
```

## Function Literals

A function literal is a reference to a (already defined) function. Section 2.8 (p. 33) explains function references.

```
FunctionLiteral := '&' [ModulePrefix] Identifier .
```

## Null Literal

The `null` literal denotes a “special” value which is different from all other values.

```
NullLiteral := null .
```

## 2.4 Variables

A variable is a storage location identified by a name. Values can be assigned to (stored in) the variable, and the value can later be retrieved by the same name.

Variable (and function) identifiers are sequences of ordinary latin letters, digits, and the underscore character.

- Identifiers must not start with a digit.
- Identifiers are case sensitive, i.e. lowercase and uppercase variants are different.
- Keywords (see appendix A.2 (p. 161)) cannot be used as identifiers.
- The maximum length of an identifier is 64 characters.

Examples for valid identifiers:

```
a
Z
AvogadroConstant
avogadro_constant
_4
x1
```

Examples for invalid identifiers:

```
9a // starts with a digit
end // is a keyword
This_identifier_is_too_long_to_be_accepted_as_it_is_over_64_chars
```

```
IdentifierChar := 'A' to 'Z' | 'a' to 'z' | '_' .
Identifier := IdentifierChar {IdentifierChar | Digit} .
```

There are two different kinds of variables:

- Global variables belong to a module (see section 2.9 (p. 34)) and exist as long as the process containing the module exists. Global variables can only be created within the module declaring them.
- Local variables belong to a function (see section 2.8 (p. 29)) and can only be referenced within their function. They are different from global variables with the same name, and exist as long as the function executes: they are created when the function is called, and are destroyed when the function returns. Hence, each invocation of a recursive function creates its own set of local variables. Function parameters are also local variables .

See section 2.9 (p. 34) for examples and an explanation of module prefixes.

```
ModulePrefix := [ModuleName | '.' ] '.' .
Variable := [ModulePrefix] Identifier .
ModuleName := Identifier .
```

## 2.5 Arrays

Arrays are collections of values. The array values can be of different type, and they can themselves again be arrays. The individual array elements are accessed by indexing with integer numbers, starting at 0 for the first element. Indexing requires putting the index value between brackets [], following the array variable.

Trying to access an element with a negative or too large index throws `ExcIndexOutOfRange`. Function `.len` (p. 50) returns the number of elements in the array.

Arrays are created by array literals, or by functions in module `array` (p. 55). An array literal is a comma-separated sequence of element values between brackets:

```
a=["One", "Two", "Three"];
print a[0] // first element
→ One
print a[2] // third element
→ Three
print len(a)
→ 3
print a[3] // there is no fourth element
→ ExcIndexOutOfRangeException thrown
```

Arrays in `m` are completely dynamic, i.e. they can grow and shrink in size. Function `.append` (p. 43) appends elements to an array:

```
append(a, "Four", "Five");
print a
→ [One,Two,Three,Four,Five]
```

## Associative Arrays



Array values can also be indexed by strings (“keys”), making the arrays “associative” and facilitating many programming tasks. Setting or getting an array element via a string key is a fast operation<sup>3</sup>. Normally, keys are case sensitive, but `array.new` (p. 60) can also create arrays using case folded keys.

Unlike indexing with numbers, indexing with strings for nonexistent index values does not throw an exception:

- Getting an element for a nonexistent key returns `null`.
- Setting an element for a nonexistent key appends the element to the array.

Arrays with string keys can still be indexed using integer values.

In array literals, preceding an element value with a key and a colon adds the corresponding key:

---

<sup>3</sup>Internally, keys are organized into a dynamic hash table.

```

h=["Joe":150, "Jack":165, "William":180, "Averell":195];
print h["Jack"]
→ 165
print h["Lucky Luke"] // element does not exist
→ null
h["Lucky Luke"]=185; // element is appended
print h
→ [150,165,180,195,185]
print h[2]
→ 180

```

See also: [.append](#) (p. 43), [.keys](#) (p. 50), module [array](#) (p. 55).

```

Literal := SimpleLiteral | ArrayLiteral .
ArrayKey := Expression .
ArrayValue := Expression .
ArrayElement := [ArrayKey ':' ] ArrayValue .
ArrayLiteral := '[' [ ArrayElement {',' ArrayElement} ] ']' .
Designator := Variable { '[' Expression ']' } .

```

## 2.6 Expressions

Generally speaking, expressions define (arithmetic, bitwise, comparison, or logical) operations on (variable, literal, or function result) operands.

### Operands

In m, there are four types of operands:

- **Designators:** the operand is the value of a variable or array element, e.g. `count`, `list[i]`.
- **Function Calls:** the operand is the result of a function call, e.g. `io.read(f, 10)`, `math.sin(x)`. Functions are explained in section 2.8 (p. 29).
- **Literals:** the operand is a literal, i.e. an explicit value, e.g. `42`, `"Hello"`.
- **Expression:** the operand is an expression in parentheses, e.g. `(7.2*x)`, `(not exists[key])`.

## Operation Precedence

Each operation has a precedence defining the order in which operations are executed: as a general rule, arithmetic and bitwise operations are executed before comparisons, and comparisons are executed before boolean operations. Within each group, multiplicative operations have higher precedence than additive ones. Operations of equal precedence are executed from left to right. The order of execution can be changed by grouping subexpressions into parentheses.

```
t=3; s=7; m="aha";
print t + 5*s - 2/4 // multiplicative before additive
→ 37.5
print s&4 > t|4 // bitwise before comparison
→ false
print 13>s or m>"b" // comparison before boolean ops
→ true
print (20+t)*(s-24) // parentheses change the order
→ -391
```

## Arithmetic Operators

The arithmetic operators are (P is the precedence):

Op	P	Description
x+y	4	Addition.
x-y	4	Subtraction.
x*y	5	Multiplication.
x/y	5	Division.
x%y	5	Integer remainder: <code>x - y*trunc(x/y)</code> ; if <code>y=0</code> , throws <code>ExcDivideByZero</code> .
-x	6	Change sign of x.



```
print 22 / 7
→ 3.142857149
print 97 % 11
→ 9
print 97 % -11
→ 9
print -97 % 11
→ -9
```

Except for %, these operations never throw an exception if an invalid operation is attempted or overflow or underflow occurs. Instead, the result becomes (negative or positive) infinity, or zero:

```
x=1e200;
print x*x
→ Inf
print -2/0
→ -Inf
print 1/x/x
→ 0
```

## Bitwise Operators

Bitwise operators work on integer numbers, treating them like binary numbers of 32 bits. Such operations are typically used to represent sets of binary states (e.g. flags) in a single value, or for hardware related operations.

The bitwise operators are (P is the precedence):

Op	P	Description
<code>x y</code>	4	Bitwise or.
<code>x^y</code>	4	Bitwise exclusive or.
<code>x&amp;y</code>	5	Bitwise and.
<code>x shl y</code>	5	Bitwise shift left.
<code>x shr y</code>	5	Bitwise shift right.
<code>~x</code>	6	Bitwise not.

```

print 1|2|4|8
→ 15
print 10&(2|4)
→ 2
print 14^11
→ 5
print ~(14&11) & (14|11) // ~(a&b) & (a|b) = a^b
→ 5
print 13 shl 4 // 13*16
→ 208
print 341 shr 2 // trunc(341 / 4)
→ 85
print ~0
→ -1

```

## Concatenation Operator

The concatenation operator concatenates two strings or a string with the string representation of another value (P is the precedence):

Op	P	Description
<code>x + y</code>	4	Concatenation: x followed by y.

Note that if neither of the two operands is a string, the two operands are assumed to be numbers and added.

```

print "One" + "Two"
→ OneTwo
print "x=" + 3/4
→ x=0.75

```

## Comparison Operators

Comparing two operands always produces a boolean value. Testing for equality and inequality works for all pairs of operands. Operands of different types (e.g. a number and a string) are never equal.

Only numbers and strings can be ordered, i.e. compared for less or greater than. Strings are ordered by their UNICODE<sup>®</sup> character values, which orders uppercase before lowercase, and does not produce a general lexical ordering.

Use `.collate` (p. 45) to lexically compare strings.

Trying to order operands other than numbers or strings throws `ExcNotComparable`.

Two arrays are only equal if they are the same array. `.equal` (p. 46) compares two arrays element by element.

`null` is only equal to itself.

The comparison operators are (P is the precedence):

Op	P	Description
<code>x = y</code>	3	true if x is equal to y.
<code>x # y</code>	3	true if x is not equal to y.
<code>x &lt;&gt; y</code>	3	The same as <code>x # y</code> .
<code>x &lt; y</code>	3	true if x is less than y.
<code>x &lt;= y</code>	3	true if x is less than or equal to y.
<code>x &gt; y</code>	3	true if x is greater than y.
<code>x &gt;= y</code>	3	true if x is greater than or equal to y.

```
print 7>5
→ true
print "o" + "ne" = "one"
→ true
print "two" < "three"
→ false
print "Two" < "three" // no lexical ordering
→ true
print 14 = "a"
→ false
print 13 # "b"
→ true
print 13 < "14"
→ ExcNotComparable thrown
```

## Boolean Operators

The boolean operators are (P is the precedence):

Op	P	Description
x or y	1	Logical or: true if either x or y is true, false if both x and y are false.
x and y	2	Logical and: true if both x and y is true, false if either x or y are false.
not x	6	Logical not: true if x is false, false if x is true.

```

print false or false, false or true, true or false,
    true or true
→ false true true true
print false and false, false and true, true and false,
    true and true
→ false false false true
print not false, not true
→ true false

```

The second operand is only evaluated if the first operand doesn't already determine the result. This is often useful when doing combined checks, as it avoids evaluation of invalid expressions:

```
ok=m#0 and 17%m = 3 // deadly 17%0 is never evaluated
```

```

Expression := Predicate {or Predicate} .
Predicate  := Comparison {and Comparison} .
Comparison :=
    Sum [ ('=' | '<>' | '#' | '<' | '>' | '<=' | '>=') Sum ] .
Sum        := Product { ('+' | '-' | '|' | '^') Product } .
Product    := Factor { ('*' | '/' | '%' | '&' | shl | shr) Factor } .
Factor     := [ '-' | '~' | not ]
            (Designator | FunctionCall | Literal | (' Expression ') ) .

```

## 2.7 Statements

Statements are the smallest unit of execution in m. Statements change values of variables, call functions and control the flow of execution. Most of the time, several statements are executed in a sequence, one after the other.



A sequence of statements is called a *statement list*. Within a statement list, statements must be separated by a semicolon. This is the only place where m requires a semicolon. In particular, there is no need to put a semicolon at the

end of each statement<sup>4</sup>. However, ending or preceding each statement with a semicolon is not an error, it just produces empty statements which are ignored during execution.

For instance, the following two code fragments are completely equivalent:

```
use math;
function f(x);
    return x*x*math.exp(x/40);
end;
for x=0 to 10 by 0.1 do;
    y=f(x);
    print x,y;
end;
```

```
use math
function f(x)
    return x*x*math.exp(x/40)
end
for x=0 to 10 by 0.1 do
    y=f(x); // this is the only required semicolon
    print x,y
end
```

```
Statement := |
    Assignment | ConstAssignment | Increment | Expression |
    IfStatement | WhileStatement | DoStatement | ForStatement |
    BreakStatement | ReturnStatement | ThrowStatement |
    TryStatement | PrintStatement .
StatementList = Statement { ';' Statement } .
```

### 2.7.1 Assignment

This statement type assigns the value of an expression to a variable or array element. It also defines the variable if it didn't occur in the preceding code yet. A variable can be reassigned as often as required, also with values of different types (although this is generally not considered good programming practice).

<sup>4</sup>This minimalistic approach was chosen to reduce the number of control characters required for a valid m script.

```
x = 28*3;
x = ["a", "b", "c"];
x[1] = "b2";
x[2] = null;
x["new"] = "d";
print x
→ [a,b,null,d]
```

When assigning an array, the array is not copied: the expression and the variable or array element it is assigned to will denote the same array:

```
ma = ["Ma", "Dalton"];
joe = ma; // joe and ma refer to the same array
joe[0] = ["Joe"]; // this also modified ma
print ma
→ [Joe,Dalton]
```

[array.copy](#) (p. 56) copies an array element by element.

If a variable is being defined, i.e. didn't occur in the preceding code, it can be marked as constant by prefixing the assignment with `const`. Array elements cannot be marked constant: a constant array can be modified after it has been assigned to another variable.

```
const C = 2.997e8;
C = 4; // illegal
const A = [1, 2, 3];
A[1] = 7; // illegal
b = A;
b[1] = 7; // perfectly legal, also modifies A[1]
print A
→ [1,7,3]
```

```
Assignment := Designator '=' Expression .
ConstAssignment := const Variable '=' Expression .
```

## 2.7.2 Increment

This statement type increments or decrements a numeric variable by a numeric expression (`+=`, `-=`), or simply by one (`++`, `--`). These statements are

just shorthand notations for full assignments<sup>5</sup>:

Increment	Equivalent Assignment
<code>x += expr</code>	<code>x = x + expr</code>
<code>x -= expr</code>	<code>x = x - expr</code>
<code>x++</code>	<code>x = x + 1</code>
<code>x--</code>	<code>x = x - 1</code>

```
s=7;
s+=13;
s--;
print s
→ 19
```

Increment := Designator  
 ('+=' Expression | '-=' Expression | '++' | '--') .

### 2.7.3 If Statement

An if statement executes some code depending on the value of boolean expressions (e.g. comparisons). Its simplest form executes statements (the print in the example) if a condition (`a > 13`) evaluates to true:

```
a=15;
if a > 13 then
  print a + " is greater than 13"
end
→ 15 is greater than 13
```

An optional else block may contain statements which are executed if the condition evaluates to false:

```
a=9;
if a > 13 then
  print a + " is greater than 13"
else
  print a + " is less than 13"
end
→ 9 is less than 13
```

<sup>5</sup>They are not completely equivalent: in `s[f(x)]+=3`, `f(x)` is evaluated only once, whereas in `s[f(x)]=s[f(x)]+3`, `f(x)` is evaluated twice.

To test for more than just two alternatives, an arbitrary number of `elsif` blocks can be added. These must occur after the `if/then` and before the (optional) `else` block:

```
a=13;
if a > 14 then
  print a + " is greater than 14"
elsif a < 13 then
  print a + " is less than 13"
elsif a = 13 then
  print a + " is equal to 13"
else
  print a + " must be 14"
end
→ 13 is equal to 13
```

If any of the conditions evaluates to `true`, the remaining conditions are not evaluated.

Throws `ExcNotBoolean` if any of the evaluated conditions is not boolean.

```
IfStatement := if Expression then StatementList
{elsif Expression then StatementList}
[else StatementList]
end .
```

## 2.7.4 While Statement

The `while` statement repeats some code as long as a condition evaluates to `true`. The condition is tested *before* each repetition.

```
a=[430, 241, 187, 53, -1, 17]; s=0;
while i<len(a) and a[i]>=0 do
  s += a[i]; i++
end;
print i, s
→ 4 911
```

Throws `ExcNotBoolean` if the condition is not boolean.

```
WhileStatement := while Expression do StatementList end .
```



### 2.7.5 Do-Until Statement

The `do` statement repeats some code until a condition evaluates to `true`. The condition is tested *after* each repetition.

```
x=2; y=x;
do
  y0=y; y=(y+x/y)/2
until y>=y0;
print y, y*y
→ 1.4142135624 2
```

Throws `ExcNotBoolean` if the condition is not boolean.

```
DoStatement := do StatementList until Expression .
```

### 2.7.6 For Statement

The `for` statement lets an index variable iterate through a range of numbers or through the elements of an array, and executes some code for each value. The index variable must be a simple variable, either local in the current function or global in the current module. It cannot be an array element or a variable in another module.

- The `for` loop iterating through a range of numbers looks as follows:

```
for index=StartExpr to EndExpr [by IncrExpr] do
  statements
end
```

The range is defined by `StartExpr` and `EndExpr`, and an optional `IncrExpr` defining the amount by which the variable is incremented after each iteration. `IncrExpr` defaults to 1.

All three expressions are evaluated only once, before the loop is entered.

The loop exits if `index > EndExpr` (if `IncrExpr > 0`), or if `index < EndExpr` (if `IncrExpr <= 0`).

```

for x=5 to 6 by 0.25 do
  print x*x
end
→ 25
   27.5625
   30.25
   33.0625
   36

```

A `for` loop over a range is equivalent to the following `while` loop:

```

index=StartExpr; e=EndExpr; d=IncrExpr;
while d>0 and index <= e or d<=0 and index >= e do
  statements;
  index += d
end

```



Care must be taken when using `for` loops with fractional numbers: rounding errors may lead to surprising results:

```

for i=5 to 6 by 0.2 do
  print i
end
→ 5
   5.2
   5.4
   5.6
   5.8
print i-6
→ 8.881784E-16

```

- The `for` loop iterating through the elements of an array looks as follows:

```

for index in ArrayExpr do
  statements
end

```

The array is defined by `ArrayExpr`. `index` iterates through all elements of the array, starting at index 0 and ending with the last element (`index len(ArrayExpr)-1`).

```

a=[430, 241, 187, 53, -1, 17]; s=0;
for x in a do
    s += x
end;
print s
→ 927

```

A for loop over an array is equivalent to the following while loop:

```

a=ArrayExpr; i=0;
while i<len(a) do
    index = a[i];
    statements;
    i++
end

```

```

ForStatement := for IndexVariable
    ( = Expression to Expression [by Expression] | in Expression )
    do StatementList end .
IndexVariable := Identifier .

```

### 2.7.7 Case Statement

The case statement executes a sequence of statements depending on the value of an expression matching the tag or tags of this sequence. It looks as follows:

```

case Expression
    in TagExpr1:
        statements1
    in TagExpr2a, TagExpr2b:
        statements2
    else
        statements3
end

```

This case statement is equivalent to the following if statement:

```

x=Expression;
if x=TagExpr1 then
    statements1
elseif x=TagExpr2a or x=TagExpr2b then
    statements2
else
    statements3
end

```

*Expression* is evaluated only once.

The tags (*TagExpr1*, *TagExpr2a*,...) are evaluated when they are tested. Once a matching tag has been found, the remaining tags are not evaluated.

Equality of expression and tag is tested using the = operator (see section 2.6 (p. 16)). Arrays are thus not compared elementwise, and string comparison is case sensitive.

The following example prints a different message for different values of *i*:

```

for i=1 to 10 do
    case i
        in 1:
            print i,"is somewhat prime"
        in 2, 3, 5, 7:
            print i,"is prime"
        else
            print i,"is not prime"
        end
    end
end
→ 1 is somewhat prime
   2 is prime
   3 is prime
   4 is not prime
   5 is prime
   6 is not prime
   7 is prime
   8 is not prime
   9 is not prime
  10 is not prime

```

```

| CaseStatement := case Expression
| { in TagList ":" StatementList }
| [ else StatementList ] end .
| TagList := Expression { ",", Expression } .

```

## 2.7.8 Break Statement

The `break` statement exits from the loop (`while`, `do-until`, `for`) containing it, and continues execution after the end of the loop.

```

x=-3;
while true do
  if x<0 then break end;
  y=x; x=x/2+1/x;
  if x>=y then break end
end;
print x, x*x
→ -3 9

```

`break` always exits the innermost loop containing it. Breaking out of an outer loop is not possible.

```

| BreakStatement := break .

```

## 2.7.9 Return Statement

The `return` statement returns the value of an expression as a function result. Outside a function, it ends execution of the module's body; the return value is discarded.

See section 2.8 (p. 29) for examples.

```

| ReturnStatement := return Expression .

```

## 2.7.10 print Statement

The `print` statement provides a simple way of producing output. It writes a line with zero, one or several expressions to the console. The expressions are separated by single spaces. `print` without expressions just outputs a new

line.

```
print "odd:", 3/7
→ odd: 0.4285714286
print
→
```

Expressions are formatted depending on their type:

- A *Number* is printed as string of length 12 or less (and rounded, if necessary). If the value cannot be represented within 12 characters, scientific representation is chosen.

```
print 13.5
→ 13.5
print 3e11
→ 300000000000
print -3e11; // -300000000000 has 13 characters
→ -3E+11
```

- A *String* is printed as is.

```
print "Hello,", 'world!'
→ Hello, world!
```

- A *Boolean* is printed as "true" or "false":

```
print 1 < 3
→ true
```

- An *Array* is printed elementwise, up to a length of 128. Elements which are themselves arrays are printed as `Array<len>`.

```

a=[];
for i=1 to 10 do append(a, i) end;
print a
→ [1,2,3,4,5,6,7,8,9,10]
a[0]=a;
print a
→ [Array<10>,2,3,4,5,6,7,8,9,10]
for i=11 to 100 do append(a, i) end;
print a
→ [Array<100>,2,3,4,5,6,7,8,9,10,11,12,13,14,
    15,16,17,18,19,20,21,22,23,24,25,26,27,28,
    29,30,31,32,33,34,35,36,37,...<100>]

```

- A *Function Reference* is printed as `&func<mod, func>`, and as such of little interest to the user (`mod` and `func` are indices into internal module and function tables).
- The *Null* value is printed as `null`.
- A *Native Object* is printed as `type@address`. `type` defines the object type, `address` is the location of the underlying native object in memory.

```

f=io.create("sample.xml");
print f
→ stream@41255c

```

For finer control over output formatting, see `.str` (p. 53) and module `io` (p. 111).

```

| PrintStatement := 'print' [ Expression { ',' Expression } ] .

```

## 2.8 Functions

Functions are a way to write repeatedly occurring computations only once, but use them wherever needed. They also help in structuring longer scripts into smaller, easily understandable units. By putting often occurring functions into separate modules (see section 2.9 (p. 34)), function libraries can be created.

Functions normally have a set of *parameters* as input and return a single *function result* as output. Since the function result can be an array, an arbitrary number of values can be returned.

The function is left by returning a value with a `return` statement. If the function is left by reaching its end, `null` is returned.

The following example declares a function `sqrt` computing the square root of a number `x` greater than or equal to 1, then calls it with parameters `x=2` and `x=9`:

```
function sqrt(x)
  y=x;
  do
    y0=y; y=(y+x/y)/2
  until y>=y0;
  return y
end // of function sqrt

print sqrt(2), sqrt(9)
→ 1.4142135624 3
```

This is quite a simple function with a single parameter `x` and a simple function result (the value of `y`).

Multiple parameters are separated by commas. The following function `find` finds the index of the first element in an array `a` with a value equal to `x` (there is a standard function for this: `array.index` (p. 57)).

```
function find(a, x)
  i=0;
  while i<len(a) and a[i]#x do
    i++
  end;
  return i
end

print find([9, 11, 13], 11)
→ 1
print find([9, 11, 13], 8)
→ 3
```

A function can be recursive, i.e. can call itself: the following function `clone`



returns a full copy of its parameter `t`. It uses `array.copy` (p. 56) to copy all the elements, and `.isarray` (p. 48) to test whether `t` is an array.

```
function clone(t)
  if isarray(t) then
    c=array.copy(t);
    // recursively clone the elements
    for i=0 to len(t)-1 do c[i] = clone(t[i]) end;
    return c
  else
    return t
  end
end
```

If a function returns an array, the call can be followed by expressions in brackets accessing certain elements:

```
a=['one':1, 'two':2, 'three':3];
print keys(a)[2]
→ three
```

## Optional Parameters

Optional parameters are parameters with a default value: if the parameter is omitted, the default value is assumed. The expression to compute the default value can be any expression which is valid in the global context (i.e. it cannot use a preceding function parameter). It is evaluated when the function is called, not when the function is declared.

When calling a function, the number of actual parameters must not be less than the number of mandatory parameters in the declaration of the function, and not be greater than the total number of declared parameters.

The following rewrites function `find` by adding an optional parameter `start` indicating the position to start searching at. The default value of `start` is 0, so calling `find` with only two parameters produces exactly the same result as before:

```
function find(a, x, start=0)
  while start<len(a) and a[start]#x do
    start++
  end;
  return start
end

print find([9, 11, 13], 11)
→ 1
print find([9, 11, 13], 11, 2) // start=2
→ 3
```

Functions with optional parameters can have options for simplified syntax in interactive use (see section 4.1 (p. 149)). Options are simply single character names for optional parameters.

```
function grow(years,interest=2) /i:interest
  a=1;
  while years>0 do
    a+=a*interest/100; years--
  end;
  return a
end

grow(10,5)
→ 1.21899442
grow/i=5 10 // works only in interactive shells
→ 1.6288946268
```

## Forward Declaration

Functions must be declared before they can be used. This means that if two functions call each other, at least one must be declared with `forward` and implemented later. In the following example, either `f` or `g` must be forward declared, since function `f` calls function `g` and vice versa:

```
function g(x, a=3.2) forward // g is made known

function f(x)
  if x<3 then
    return g(x*x) // g is called
  else
    return x+2
  end
end

function g(x, a=3.2) // here g is declared
  return f(x+a)
end
```

The default values of the optional parameters of the forward declared function are only used to mark optional parameters, their values are ignored. The default values are taken from the function implementation. The number of mandatory and optional parameters in forward declaration and implementation must match.

## Function References

Function references allow to change the function called in an expression during the execution of a script: when a function reference is assigned to a variable or a parameter, the function can be called via the variable. The reference of a function is obtained by prefixing it with an ampersand character &:

```
f=&lower; // f now references the lower function
print f("Hello") // a call to lower
→ hello
f=&upper; // f now references the upper function
print f("Hello") // a call to upper
→ HELLO
```

Function references are often used to pass a function as a parameter to another function: the function `integ` approximates the integral of `f` from `a` to `b`:

```

function integ(f, a, b, n=100)
  s=(f(a)+f(b))/2; h=(b-a)/n;
  for i=1 to n-1 do
    s+=f(a+i*h)
  end;
  return s*h
end

function inv(x) return 1/x end
print integ(&inv, 1, 2)
→ 0.6931534305
print integ(&math.sin, 0, math.pi/2)
→ 0.9999794382
print integ(&math.sin, 0, math.pi/2, 10000)
→ 0.9999999979

```

```

FunctionDeclaration := function Identifier '(' [ParameterList] ')'
  ( forward | {FunctionOption} StatementList end ) .
ParameterList := (MandatoryParameter {' ' MandatoryParameter} |
  OptionalParameter) {' ' OptionalParameter} .
MandatoryParameter := Identifier .
OptionalParameter := Identifier '=' Expression .
FunctionOption := '/' OptionName ':' ParameterName .
OptionName := IdentifierChar | Digit .
ParameterName := Identifier .

ActualParameterList := Expression {' ' Expression} .
FunctionCall :=
  [ModulePrefix] Identifier '(' [ActualParameterList] ')'
  { '[' Expression ']' } .

```

## 2.9 Modules

A module in `m` is a script (a text file) which can be loaded by other scripts, giving access to the functions and variables defined in the module.

Modules serve two purposes:

- They help in structuring complex scripts and make them easier to understand and maintain.
- They offer a way of extending the functionality of `m` by adding new functions which can then be used by all scripts or interactive `m` sessions. Entire libraries of often needed functions can be created that

way. The standard library of `m` described in the next chapter is organized into modules.

To load a module, a `use` clause is required:

```
use ModuleName1, ModuleName2, ...
```

This loads the modules `ModuleName1`, `ModuleName2` and so on, and initializes them, i.e. executes their main code. Each module is only initialized once per process, even if it is loaded several times by different modules.

Module names are not case sensitive, since they are related to file names on the underlying operating system.



```
use System // load module "system"
print System.appdir; // this will work
print system.appdir // this is the same
```

An alias name can be used in addition to the module name to denote the module, e.g. to abbreviate a long module name. Alias names are local to the module containing the `use` clause. Like module names, they are not case sensitive:

```
use ModuleName as AliasName
```

As an example, consider the following module `accounts` maintaining a list of accounts and allowing transfers between them:

```
S=[]; // initialization of the module
function get(nr)
  x=accounts.S[nr];
  // all accounts start at zero
  if x=null then x=0 end;
  return x
end
function xfer(f, t, x)
  ..S[f]=get(f)-x;
  ..S[t]=get(t)+x
end
```

Within the functions, the global variable `S` must be prefixed by the module name (`accounts.S`), or by the double dot prefix indicating the current module (`..S`).

The module can then be used as follows:

```
// load the module and name it 'acc'.
use accounts as acc
// transfer money out of the blue to the bank
acc.xfer('blue', 'bank', 100000);
print acc.get('bank')
→ 100000
// transfer money from the bank to the Daltons
acc.xfer('bank', 'Daltons', 10000);
print acc.get('bank')
→ 90000
// show all accounts
print acc.S
→ [-100000, 90000, 100000]
```

## Module Prefixes

Global variables and functions must normally be prefixed by the name of the module defining them (or the corresponding alias), and a dot. The prefix for the main script and the builtin functions and variables is just a dot, without a name.

Within a module, global variables and functions of the same module can be prefixed by a *double dot* `..`: in the code for module `accounts` above, `..S` denotes the same variable as `accounts.S`.

The prefixing is not always required when the variable or function is referenced in the module containing it. Furthermore, functions defined in the main script or builtin standard functions only need a prefix if a function with the same name exists in the current module. The following table summarizes how variables and functions *without module prefix* are interpreted:



	Variable $x$	Function $f$
main module	global $.x$	$.f$
function in main module	local $x$	$.f$
module $M$	global $M.x$	$M.f$ if it exists, $.f$ otherwise
function in module $M$	local $x$	$M.f$ if it exists, $.f$ otherwise

## Module Version

Each module has a version, which is a number in the form `major.minor`; the minor component by convention has a 1/100th granularity.

The module version is a special variable `version`, which can only be modified in the module itself by assigning a number literal to it. If no number has been assigned, the version is `0.0`. The version of an incorrectly loaded optional module (see below) is `null`.

Source of module `client`:

```
version=1.23
```

```
use client
// require at least version 1.20 of client module
if client.version>=1.20 then
  ...
end
```

The version of the builtin module is always the version of the `m` application. See also `.version` (p. 55).

## Optional Modules

Not all devices support all modules, or a module may simply not be installed on a device. To cope with these cases in the code, a module can be loaded in a `use` clause with the `try` prefix:

```
use try ModuleName
```

Loading a module with the `try` prefix has the following effects:

- If the module and all the modules it uses are correctly loaded, the result is almost the same as without `try`. However, a reference to an undefined function or variable of the module will not be detected until the code reaches the corresponding statement and throws `ErrNotAvailable`. This allows to run code even if some functions or variables of a module do not exist.
- If the module `ModuleName` itself or one of the modules it uses is not

found or cannot be loaded, no error is marked. However, all references to its variables and functions will result in `ErrNotAvailable` being thrown; only the module's `version` variable is accessible and will return `null`.

```
use try nirvana
nirvana.f(1, 2)
→ ErrNotAvailable thrown
print nirvana.val
→ ErrNotAvailable thrown
// inirvana.version is null: the module cannot be used
print nirvana.version
→ null

use try math
// there is no sinh function in module math
print math.sinh(1.2)
→ ErrNotAvailable thrown
// math.version is not null: the module can be used
print math.version
→ 1.08
```

```
ModuleImportList := use ModuleImport { ',' ModuleImport } .
ModuleImport := [try] ModuleName [as AliasName] .
AliasName := Identifier .
```

## 2.10 Exceptions

An exception is the result of an attempt to perform an invalid operation. By default, exceptions result in a popup window showing the exception message text.

An exception thrown by `m` will always have the following format:

```
ExceptionFormat := tag ':' message
```

The tag is always an (english) identifier, and independent of the language chosen when installing `m`. The message however depends on the language. See section A.1 (p. 157) for a list of `m` exception tags.



## Catching Exceptions

Exceptions can also be handled (“caught”) in `m` itself:

```
try
  // code potentially throwing exceptions
catch exc by
  // code handling the exception exc
end
```

The result of such a `try` block is the following:

- If the code between `try` and `catch` does not throw any exception, the code between `catch` and `end` will never be executed.
- If the code between `try` and `catch` does throw an exception, the exception will be assigned to the variable denoted by `catch` and the following code will be executed.

In the following example, `a[1]` tries to access a non-existing element. `m` throws an `ExcIndexOutOfRange`, which is caught and simply printed:

```
try
  a=[12];
  print a[1]
catch e by
  print "Got", e
end
→ Got ExcIndexOutOfRange: Array index is out of range
```

Try blocks can be nested to any depth (as long as the required memory is available).

```
TryStatement := try StatementList
               catch ExceptionVariable by StatementList end .
ExceptionVariable := Identifier .
```

## Throwing Exceptions

Exceptions can also be thrown explicitly in the code:

```
throw expression
```

This will evaluate `expression` and use it as exception message. In the following example, an exception with the message “state.dat does not exist” will be thrown if this file does not exist.

```
if not files.exists("state.dat") then
  throw "state.dat does not exist"
end
```

```
| ThrowStatement := throw Expression .
```

## 2.11 Source Structure

After introducing all elements of the `m` language, the complete structure of an `m` source can be defined:

```
| MSource := {ModuleImport | FunctionDeclaration | StatementList} .
```

The `StatementLists` (there can be several) are the “main code” of the script which is executed directly. In a module, this corresponds to the module initialization code which is executed the first time the module occurs in a `use` clause.

## 3. Library

The `m` library contains a large number of functions, organized into modules. Some functions are the standard functions you expect in any serious programming language. Others are very specific to the typical capabilities of a smart phone.

The following sections describe the modules which are always available. New modules can be added by yourself or by a third party, either written in `m`, or written for the native platform. For Symbian OS, this is typically a dynamic library.

See section 2.9 (p. 34) for more information on using and writing modules. Just a reminder: to use any of the standard modules, you have to load it via the `use` clause:

```
use math
print math.random()
→ 0.1488330803
```

### 3.1 Path and File Names

A complete file name in `m` (and in Symbian OS) consists of a drive, a directory path, and the file name with extension. The drive is followed by a colon; drive, directories and file name are separated by backslashes (`\`). Since the backslash is also the escape character in strings, each backslash must be entered as two backslashes (unless simplified interactive syntax is used, see section 4.1 (p. 149)):

```
path="c:\\documents\\mShell\\script.m";
```

By convention, a directory name always ends with a backslash, allowing immediate differentiation between directory names and file names.

To avoid the need for a fully specified file name, each process in `m` maintains

a current directory (see [.cd](#) (p. 43)). Unlike in DOS/Windows, which maintains a current directory for each drive, there is only one current directory in `m`, which always includes the drive.

All functions taking file or directory names as arguments therefore accept absolute, drive-relative or relative file names:

- Absolute file names start with the drive letter. The directory path always starts from the root of the drive, even if the first backslash is missing.

```
cd("c:documents");  
print cd()  
→ c:\documents\
```

- Drive-relative file names start with a backslash. They are always relative to the root of the current drive (which is part of the current directory).

```
cd("\\documents");  
print cd()  
→ c:\documents\
```

- Relative file names start with a directory name, or simply a file name. They are always relative to the current directory.

```
cd("mShell");  
print cd()  
→ c:\documents\mShell\
```

`m` also interprets two special directory names:

- A single dot refers to the current directory.
- A single dot refers to the preceding directory.

Single and double dots can occur anywhere in the directory path.

```
cd("c:\\documents");  
cd(".\\mShell"); // . refers to c:\documents  
print cd()  
→ c:\documents\mShell\  
cd("../Jotter"); // .. refers to c:\documents  
print cd()  
→ c:\documents\Jotter\  

```

## 3.2 Builtin Functions and Constants

The functions listed here are the standard `m` functions available without importing any module. They can be called without a module or alias prefix, or with an empty prefix (a dot).

```
print date();  
print .date()
```

Both statements have the same effect.

### `.append`

- function `append(array, element, ...)` → null

Append one or more elements to the the end of `array`. The length of `array` is increased by the number of elements appended.

```
arr=[];  
append(arr, 17, "x");  
print arr  
→ [17,x]
```

### `.cd`

- function `cd()` → String
- function `cd(newpath)` → String

Gets and sets the current (default) directory. This is the directory all file or directory operations relate to. See also section 3.1 (p. 41).

Without an argument, `cd` returns the current directory without modifying it. With a single argument, it changes the current directory to `newpath` and returns the previously set current directory. `newpath` can be absolute, or relative to the current directory.

```
cd("c:\\");  
print cd("system")  
→ c:\\  
print cd("apps")  
→ c:\\system\\  
print cd()  
→ c:\\system\\apps\\
```

See also: [files.mkdir](#) (p. 82), [files.rmdir](#) (p. 83)

## .char

- function `char(array) → String`

Converts the array of numbers `array` to a string, interpreting each number as a UNICODE® BMP character code. The codes must be numbers between 0 and  $2^{16} - 1 = 65535$ .

```
print char([72,101,108,108,111])  
→ Hello
```

See also: [.code](#) (p. 45)

## .cls

- function `cls() → null`

Clears the screen, deleting all console output produced so far.

```
cls()
```

### `.code`

- `function code(text) → Array`
- `function code(text, pos) → Number`

With a single argument, converts `text` to an array containing the UNICODE<sup>®</sup> number for each character. With two arguments, returns the code for the character at position `pos` of `text`.

```
print code("Hello")
→ [72,101,108,108,111]
print code("Hello", 1)
→ 101
```

See also: `.char` (p. 44).

### `.collate`

- `function collate(s1, s2) → Number`

Compare the two strings `s1` and `s2`, correctly ordering accents and umlauts depending on the current locale. Returns a negative number if `s1 < s2`, zero if `s1 = s2`, a positive number if `s1 > s2`.

```
// Flüge comes before Flugzeug in lexical ordering
print collate("Flüge", "Flugzeug")
→ -1
// simple raw ordering produces the wrong result
print "Flüge" < "Flugzeug"
→ false
```

See also: constant `array.collate` (p. 63).

### `.date`

- `function date() → String`

Get the current local date and time in the format `YYYY-MM-DD hh:mm:ss`. See also module `time` (p. 131).

```
print date()
→ 2005-02-21 12:18:55
```

## **.equal**

- `function equal(a, b) → Boolean`

Compares two values `a` and `b` for equality and returns `true` if they are equal, `false` if they are not equal. Unlike the `m` language `=` operator, this function compares arrays elementwise: two arrays are identical if they have the same length and all their elements are equal.

```
a=[1, 2, [3, 4]]
b=a;
print a=b, equal(a, b)
→ true true
b=[1, 2, [3, 4]];
print a=b, equal(a, b)
→ false true
```

Note that the function will crash `m` if you pass two identical recursive arrays for which equality or inequality cannot be determined.

```
a=[0]; a[0]=a;
b=[0]; b[0]=b;
equal(a, b) // this will crash m
```

## **.delete**

- `function delete(text, start) → String`
- `function delete(text, start, length) → String`

Deletes the substring from `text` from position `start`, either to the end of `text`, or the next `length` characters. The first character has position 0.

Throws `ExcStringPosOutOfRange` if not `0 ≤ start ≤ len(text)`, or if not `0 ≤ length ≤ len(text) - start`.

```
print delete("Hello world!", 6)
→ Hello
print substr("Hello world!", 3, 5)
→ Helrld!
```

See also: [.substr](#) (p. 54)



### **.hexnum**

- function `hexnum(text) → Number`

Converts the string `text` representing a hexadecimal integer value into the value. The value can be signed. Uppercase and lowercase digits are allowed, and leading and trailing blanks are ignored.

```
print hexnum("1fff");  
→ 8191  
print hexnum(" -ABACADA ");  
→ -180013786
```

See also: [.num](#) (p. 51)

### **.hexstr**

- function `hexstr(number, width=0) → String`

Formats `number` into an integer hexadecimal value. If necessary, zeros are added *before* the string until its length is at least `width`.

```
print hexstr(8191)  
→ 1fff  
print hexstr(-180013786, 12)  
→ -0000abacada
```

See also: [.str](#) (p. 53)

### **.index**

- function `index(text, pattern, start=0, folded=false) → Number`

Searches the string `text` for the first occurrence of the string `pattern` at or after `start` and returns the position. If `pattern` does not occur, -1 is returned. If `folded=true`, the comparison between `text` and `pattern` ignores case.

Throws `ExcStringPosOutOfRange` if not `0 <= start <= len(text)`.

```
print index("To be, or not to be", "to be")
→ 14
print index("To be, or not to be", "to be", 0, true)
→ 0
print index("To be, or not to be", "to be", 1, true)
→ 14
print index("To be, or not to be", "to be or not")
→ -1
```

See also: [.rindex](#) (p. 51)

### **.isarray**

- function isarray(expression) → Boolean

Returns true if expression is an array, false if it is any other type.

```
print isarray([])
→ true
print isarray("String")
→ false
```

### **.isboolean**

- function isboolean(expression) → Boolean

Returns true if expression is a boolean (i.e. true or false), false if it is any other type.

```
print isboolean(4 > 5)
→ true
print isboolean(4+5)
→ false
```

### **.isFunction**

- function isfunction(expression) → Boolean

Returns true if expression is a function reference, false if it is any other type.

```
print isfunction(&cd)
→ true
print isfunction(cd())
→ false
```

### **.isnative**

- function isnative(expression) → Boolean

Returns true if expression is a native object, false if it is any other type.

```
print isnative(io.create("sample.xml"))
→ true
print isnative([])
→ false
```

### **.isnum**

- function isnum(expression) → Boolean

Returns true if expression is a number, false if it is any other type.

```
print isnum(13.26)
→ true
print isnum("13.26")
→ false
print isnum(num("13.26"))
→ true
```

### **.isstr**

- function isstr(expression) → Boolean

Returns true if expression is a string, false if it is any other type.

```
print isstr("Hello")
→ true
print isstr(null)
→ false
```

### **.keys**

- `function keys(array) → Array`

Returns an array of length `len(array)`, with each element set to the string key of the element at this position in `array`, or set to `null` if the element at this position has no key.

```
a=["one":1, "two":2, 3, "four":4, 5];
print keys(a)
→ ["one", "two", null, "four", null]
```

### **.len**

- `function len(array) → Integer`
- `function len(text) → Integer`

Returns the length (number of elements) of the array `array`, or the length (number of characters) of the string `text`.

```
print len("Hello")
→ 5
print len("")
→ 0
print len([7, 8, 9])
→ 3
print len([])
→ 0
```

### **.lower**

- `function lower(text) → String`

Returns a copy of `text`, with all uppercase characters converted to their lowercase equivalent.

```
print lower("Hello")
→ hello
print lower("WATCH OUT!")
→ watch out!
```

### `.num`

- `function num(text) → Number`

Converts the string `text` representing a numeric value into the value. The syntax for the number is the same as for numeric literals (see 2.3 (p. 7)). Leading and trailing blanks are ignored.

```
print 21+num('21')
→ 42
print num(" -15.8e4 ")
→ -158000
```

### `.replace`

- `function replace(text, old, new) → String`

Replaces all occurrences of `old` in `text` by `new`, and returns the string with replacements made. `old` and `new` need not have the same length.

```
print replace("Hello world!", "l", "ll")
→ Helllllo worlld!
print replace("Hello world!", "l", "")
→ Heo word!"
```

### `.rindex`

- `function rindex(text, pattern, start=len(text)-1, folded=false) → Number`

Searches the string `text` for the *last* occurrence of the string `text` at or *before* `start` and returns the position. If `pattern` does not occur, `-1` is returned. If `folded=true`, the comparison between `text` and `pattern` ignores case.

Throws `ExcStringPosOutOfRange` if not `-1 <= start < len(text)`.

```
print rindex("To be, or not to be", "To be")
→ 0
print rindex("To be, or not to be", "To be", 18, true)
→ 14
print rindex("To be, or not to be", "To be", 13, true)
→ 0
print rindex("To be, or not to be", "to be or not")
→ -1
```

See also: [.index](#) (p. 47)

## **.sleep**

- function `sleep(microseconds)` → null

Pauses execution for (at least) the number of microseconds (1/1000 of a second) before returning. If `microseconds` is negative or zero, execution continues immediately, but other `m` processes immediately get a chance to run, before they are preempted by the scheduler.

```
sleep(500) // wait for 1/2 s
```

## **.split**

- function `split(text)` → Array
- function `split(text, separator)` → Array

With one argument, splits `text` into words separated by any amount of white space<sup>1</sup>.

With two arguments, splits `text` into substrings at each occurrence of `separator`. `separator` can be of any length.

```
print split(" To be, or not to be?")
→ [To,be,,or,not,to,be?]
print split("Line 1<BR><BR><B>Line 3</B><BR>", "<BR>")
→ [Line 1,,<B>Line 3</B>,]
```

---

<sup>1</sup>White space: a sequence of characters equal to or less than space. This includes tab and newline.

## .str

- function `str(expression, width=0) → String`
- function `str(number, width, decimals) → String`

Converts an expression or a number to string:

- The first form converts an expression to a string, using the same rules as the `print` statement (see 2.7.10 (p. 27)):

```
print str(1 < 3)
→ true
```

- If `width ≥ 0`, spaces are added *before* the string until its length is at least `width`. The result is thus right adjusted.

```
print str(1 < 3, 8)
→      true
```

- If `width < 0`, spaces are added *after* the string until its length is at least `-width`. The result is thus left adjusted.

```
print str("hello", -8) + "world"
→ hello  world
```

- The second form formats `number` into a fixed or floating point representation, depending on `decimals`:
  - If `decimals = 0`, the number is represented without decimal positions and without decimal point, as if it were an integer:

```
print str(10000/7, 6, 0)
→ 1429
```

- If `decimals > 0`, the number is represented with decimal point and the given number of decimal positions:

```
print str(10000/7, 0, 3)
→ 1428.571
```

- If `decimals < 0`, the number is represented with floating point and the given number of *significant digits*:

```
print str(10000/7, 10, -3)
→ 1.43E+03
print str(10000/7, 0, -1)
→ 1E+03
```

### **.substr**

- function substr(text, start)→ String
- function substr(text, start, length)→ String

Extracts a substring from text from position start, either to the end of text, or the next length characters. The first character has position 0.

Throws `ExcStringPosOutOfRange` if not  $0 \leq \text{start} \leq \text{len}(\text{text})$ , or if not  $0 \leq \text{length} \leq \text{len}(\text{text}) - \text{start}$ .

```
print substr("Hello world!", 6)
→ world!
print substr("Hello world!", 3, 5)
→ lo wo
```

### **.trim**

- function trim(text)→ String

Returns a copy of text, with leading and trailing blanks removed.

```
print trim("Hello")
→ Hello
print trim(" world! ")
→ world!
```

### **.upper**

- function upper(text)→ String

Returns a copy of text, with all lowercase characters converted to their uppercase equivalent.



```
print lower("Hello")
→ HELLO
print lower("watch out!")
→ WATCH OUT!
```

## Constants

- `const version = 1.17`

The current version of `m`. Of course, for a different version this number will be different from 1.17.

## 3.3 Module `array`: Array Functions

This module provides utility functions to create, manipulate, search and sort arrays.

### `array.concat`

- `function concat(array1, array2, ...) → Array`

Concatenates all arguments to a single array and returns it. Any keys of the arrays are copied to the resulting array. If the same key occurs more than once, the key will reference the element where it occurred last.

```
a=[1, 2, "three":3, 4, 5];
b=[7, "eight":8];
c=array.concat(a, b, [9]);
print c, c["eight"]
→ [1,2,3,4,5,7,8,9] 8
print keys(c)
→ [null,null,three,null,null,null,eight,null]
```

## array.copy

- function copy(array, start=0) → Array
- function copy(array, start, length) → Array
- function copy(array, indices) → Array

Extracts a copy of array:

- from element start to the end of the array, or length elements,
- if indices is an array, the elements with indices in indices.

Only the array is copied, its elements remain the same (this is only relevant if the elements are themselves arrays).

Any keys of the copied elements are also copied to the new array.

Throws `ExcIndexOutOfRangeException` if not  $0 \leq \text{start} \leq \text{len}(\text{array})$ , or if not  $0 \leq \text{length} \leq \text{len}(\text{array}) - \text{start}$ , or if any  $0 \leq \text{indices}[i] < \text{len}(\text{array})$ .

```
a=[1, 2, "three":3, 4, 5];
print array.copy(a)
→ [1,2,3,4,5]
print array.copy(a, 3)
→ [4,5]
b=array.copy(a, 1, 3);
print b, b["three"]
→ [2,3,4] 3
print array.copy(a, [3, 2])
→ [4, 3]
```

## array.create

- function create(len, initval) → Array
- function create(len1, len2, ..., lenn, initval) → Array

Creates a one-dimensional array of length len, or a multi-dimensional array of arrays, with dimensions len1 x len2 x ... x lenn, with all array elements set to initval.

```
a=array.create(3,3,0); // create a 3x3 matrix of zeros
print a
→ [[0,0,0],[0,0,0],[0,0,0]]
b=array.create(10, "x"); // create an array of ten "x"
print b
→ [x,x,x,x,x,x,x,x,x,x]
```

## array.fill

- function fill(array, val, start=0) → null
- function fill(array, val, start, length) → null

Sets the elements of array `array` to `val`, from element `start` to the end of the array, or `length` elements.

Throws `ExcIndexOutOfRangeException` if not `0 <= start <= len(array)`, or if not `0 <= length <= len(array) - start`.

```
a=[1,2,3,4,5];
array.fill(a, 0);
print a
→ [0,0,0,0,0]
array.fill(a, false, 1, 2);
print a
→ [0,false,false,0,0]
```

## array.index

- function index(array, val, start=0) → Number

Searches the array `array` for the first element at or after `start` equal to `val`, and returns the index of the element. If there is no such element, returns -1. Elements are compared using the builtin function `.equal` (p. 46).

Throws `ExcIndexOutOfRangeException` if not `0 <= start <= len(array)`.

```

a=["To", "be", "or", "not", "to", "be"];
print array.index(a, "be")
→ 1
print array.index(a, "Be")
→ -1
print array.index(a, "be", 2)
→ 5
print array.index(a, "be", 6)
→ -1

```

See also: [array.rindex](#) (p. 61)

## array.insert

- function `insert(array, pos, element, ...)` → null

Inserts one or more elements into `array` before position `pos`. The elements at or after `pos` are moved up. The length of `array` is increased by the number of elements inserted.

Throws `ExcIndexOutOfRangeException` if not `0 <= pos <= len(array)`.

```

arr=[29, 18, -4];
array.insert(arr, 2, 17, "x");
print arr
→ [29,18,17,x,-4]

```

See also: [.append](#) (p. 43)

## array.isort

- function `isort(array, desc=false, mode=raw, ind=[0,1,...,len(array)-1])` → Array

Sorts the indices `ind` such that the elements `array[ind[i]]` are sorted in ascending order, or in descending order if `desc=true`, and returns the sorted indices.

String comparisons are performed according to `mode`<sup>2</sup> (one of `array.raw`, `array.fold`, `array.collate`).

---

<sup>2</sup>This sort is always stable.

Throws `ExcNotComparable` if the elements of interest in `array` are not all numbers or not all strings.

Throws `ExcIndexOutOfRange` if any element of `ind` does not properly index into `array`.

See also: [`array.sort`](#) (p. 62), [`array.copy`](#) (p. 56)

```
a=[412,-302,18,2077,22,149,18];
ind=array.isort(a);
print ind
→ [1,2,6,4,5,0,3]
print array.copy(a, ind)
→ [-302,18,18,22,149,412,2077]
print array.isort(a, true)
→ [3,0,5,4,2,6,1]
a=["To", "be", "or", "not", "to", "be"];
print array.isort(a)
→ [0,1,5,3,2,4]
print array.isort(a, false, array.fold)
→ [1,5,3,2,0,4]
print array.isort(a, false, array.fold, [1,2,3])
→ [1,3,2]
```

## `array.leindex`

- function `leindex(arr, val, mode=raw) → Number`

Searches the *sorted* array `arr` for the first index of the largest element less than or equal to `val`. Comparisons use the specified mode (one of [`array.raw`](#) (p. 63), `array.fold`, `array.collate`).

Returns `-1` if all elements of `arr` are larger than `val`.

Since the array is sorted, searching can be performed much more efficiently than with an unsorted array. The difference is however only noticable for relatively large arrays (around 100 elements or more).

```

a=[412,-302,18,2077,22,149,18,21];
array.sort(a);
print a
→ [-302,18,18,21,22,149,412,2077]
print array.leindex(a, 22)
→ 4
print array.leindex(a, 3000)
→ 8
print array.leindex(a, -3000)
→ -1
print array.leindex(a, 18)
→ 1

```

## array.new

- function new(size=0, foldedkeys=false) → Array

Creates a new array of length 0, with pre-allocated capacity for up to size elements.

For large arrays, pre-allocating the correct size is considerably more efficient. It avoids reallocating and copying the array contents, and it ensures the array being of minimal size. On the other hand, besides effects on memory needs and runtime, pre-allocating an array will never change the result of any computation in m.

If foldedkeys=true, the string keys of the array are compared folded, i.e. are not case sensitive. This is the only way of creating an associative array with keys that are not case sensitive.

```

a=array.new(1000);
for i=1 to 1000 do
  a=append(a, i) // will never allocate memory
end;
a=array.new(); // same as a=[]
print a
→ []
a=array.new(5, true); // keys of a ignore case
a["one"] = 1;
a["ONE"] = 2;
print a, keys(a)
→ [2] [one]

```

## `array.remove`

- function `remove(array, start, length=1) → null`
- function `remove(array, key) → null`

Removes one or several elements from `array`. The elements after the removed one(s) are shifted accordingly, and the length of `array` is reduced by the number of removed elements.

The first form removes a region of length `length`, starting at `start`. It throws `ExcIndexOutOfRange` if not `0 <= start <= len(array)`, or if not `0 <= length <= len(array) - start`.

The second form removes the single element with string key `key`. It throws `ExcNoSuchKey` if this key does not exist.

```
a=["one":1, "two":2, 3, "four":4, 5];
array.remove(a, 3);
print a, keys(a)
→ [1,2,3,5] [one,two,null,null]
array.remove(a, "one");
print a, keys(a)
→ [2,3,5] [two,null,null]
array.remove(a, 0, 3);
print a, keys(a)
→ [] []
```

## `array.rindex`

- function `rindex(array, val, start=len(array)-1) → Number`

Searches the array `array` for the first element at or *before* `start` equal to `val`, and returns the index of the element. If there is no such element, returns `-1`. Elements are compared using the builtin function `.equal` (p. 46).

Throws `ExcIndexOutOfRange` if not `-1 <= start < len(array)`.

```

a=["To", "be", "or", "not", "to", "be"];
print array.rindex(a, "be")
→ 5
print array.rindex(a, "Be")
→ -1
print array.rindex(a, "be", 4)
→ 1
print array.rindex(a, "be", 0)
→ -1

```

See also: [array.index](#) (p. 57)

## array.sort

- function sort(array, desc=false, mode=raw) → null

Sorts the array `array` in ascending order, or in descending order if `desc=true`. String comparisons are performed according to `mode`<sup>3</sup> (one of `array.raw`, `array.fold`, `array.collate`, see below).

Throws `ExcNotComparable` if the elements are not all numbers or not all strings.

See also: [array.isort](#) (p. 58)

```

a=[412,-302,18,2077,22,149,18];
array.sort(a);
print a
→ [-302,18,18,22,149,412,2077]
array.sort(a, true);
print a
→ [2077,412,149,22,18,18,-302]
a=["To", "be", "or", "not", "to", "be"];
array.sort(a);
print a
→ [To,be,be,not,or,to]
array.sort(a, false, array.fold);
print a
→ [be,be,not,or,To,to]

```

---

<sup>3</sup>Sorting is not stable if `mode#raw`.



## array Constants

- `const collate = 2` This mode correctly compares accents and umlauts, depending on the current locale.
- `const fold = 1` This mode ignores case when comparing.
- `const raw = 0` This mode directly compares 16-bit character codes.

## 3.4 Module `audio`: Audio Functions

This module provides audio functions: generating synthetic beeps and DTMF sequences, playing most audio file types (e.g. MP3), and recording and editing AU format and WAV format files.

To directly play an existing audio file, use `audio.play` (p. 67).

To play parts of a file or record to a file, use `audio.open` (p. 66), followed by calls to `audio.play` (p. 67), `audio.record` (p. 69) and `audio.stop` (p. 69).

Each file has a recorded length (its “duration”) and the “head position” the player is at or will start at. Both are measured in milliseconds. `audio.len` (p. 65) and `audio.pos` (p. 68) access them. `audio.cut` (p. 65) cuts a part out of a recording.

Please note: while it is possible to record phone conversations on most devices using this module, due to limitations in the underlying Symbian OS APIs, sound cannot be sent to a phone uplink. The behaviour when playing tones or sound during a phone call varies between devices; some will throw `ErrInUse`, others will simply mute the sound.

### `audio.beep`

- `function beep(hz=880, ms=800) → null`

Plays a synthetic beep with frequency `hz` Hertz for a duration of `ms` milliseconds.

This function immediately returns, before playing completes. Exceptions can therefore be thrown anywhere in the following code.

Throws `ErrInUse` if the sound unit is busy playing or recording another

sound. Throws `ExcValueOutOfRangeException` if the frequency is not positive.

```
audio.beep(440, 1000)
```

## audio.busy

- function `busy()` → Boolean

Returns `true` if the last playing function (`audio.beep`, `audio.dtmf`, `audio.play`) is still producing sound, or if sound is still being recorded (after `audio.record`). Returns `false` otherwise.

This function checks only the current `m` process: it will return `false` if the sound unit is in use by another process (inside or outside of `m`).

```
audio.beep(440, 1000);
while audio.busy() do
  io.print(io.stdout, '.'); sleep(200)
end;
print "beep ended"
→ .....beep ended
```

## audio.close

- function `close()` → null

Closes the currently accessed audio file.

Throws `ErrInUse` if the file is being played or recorded. Thus, to forcibly close a file, use:

```
audio.stop();
audio.close()
```

## `audio.cut`

- function `cut(start, end=0) → null`

Compatibility of function <code>audio.cut</code>	
Sony Ericsson phones cannot truncate at the beginning, <code>start=0</code> is mandatory.	<code>ErrNotSupported</code>

Cuts the current audio file at the beginning and/or end. The initial `start` milliseconds and the final `end` milliseconds will be removed.

Throws `ErrInUse` if the file is being played or recorded, `ErrAccessDenied` if the file has not been opened for writing, and `ErrArgument` if any of the cropped parts are outside the current file.

```
// truncate the current file by 10% on both ends
audio.cut(0.1*audio.len(), 0.1*audio.len())
```

## `audio.dtmf`

- function `dtmf(digits) → null`

Plays the string `digits` as DTMF (dual-tone multi-frequency) tones (“tone dialling”). Valid characters for digits are 0 to 9, A to D, # and \*. All other characters are ignored.

Throws `ErrInUse` if the sound unit is busy playing or recording another sound.

```
// play with ascending high frequency
audio.dtmf('147*2580369#ABCD')
```

## `audio.len`

- function `len() → Number`

Returns the length (“duration”) of the current file, in milliseconds.

Throws `ErrNotReady` if no file has been opened.

## audio.open

- function open(file, flags=0, rate=8000) → Number

*Permissions:* Read(file) / Read+Write(file)

Opens or creates a file for playing and/or recording, and returns the length of the file (“duration”) in milliseconds.

Whether the file is opened or created is determined by flags:

- const **rw** = 1 Open an existing file for recording.
- const **wav** = 2 Create a file in Microsoft’s WAV format.
- const **au** = 3 Create a file in Sun’s AU format.

When creating a file, you may combine `audio.wav` or `audio.au` with one of the following flags selecting the codec:

- const **alaw** = 0 Use A-law compression (13-bit to 8-bit) codec.
- const **mulaw** = 16 Use  $\mu$ -law compression (13-bit to 8-bit) codec.
- const **pcm8** = 32 Use 8-bit direct pulse-code modulation codec.
- const **pcm16** = 48 Use 16-bit direct pulse-code modulation codec.
- const **ima** = 64 Use IMA adaptive differential PCM codec.

To summarize: `audio.open` acts according to the following scheme:

- If flags=0 (the default), opens the file for playing. Attempts to record to it or to truncate it will throw `ErrAccessDenied`.
- If flags contains `audio.rw`, opens the file for playing and recording. Format, codec and sample rate will be taken from the existing file.
- If flags contains `audio.wav` or `audio.au`, creates a new file in WAV or AU format, and the specified codec is chosen. `rate` indicates the sample rate in Hz (samples per second). The sample rates supported depend on codec and device.

For a newly created file in WAV or AU format, the default codec is A-law, and the default sample rate is 8 kHz.

Throws `ErrInUse` if a file is already being played or recorded.

```
// Create a new file with default codec and sample rate
file='sample.wav';
audio.open(file, audio.wav);
// record sound until the file exceeds 200 kB
audio.record();
while files.size(file)<=200000 do
    sleep(1000)
end;
audio.stop();
print 'Recorded',audio.len(),'ms in ',
    files.size(file),'bytes.';
print files.size(file)/audio.len(),' kB/s'
→ Recorded 25260 ms in 202124 bytes.
→ 8.0017418844 kB/s
// play the file
audio.play(); audio.wait()
```

```
// Do the same in full lossless CD quality
audio.open(file, audio.wav | audio.pcm16, 44100);
// record sound until the file exceeds 200 kB
audio.record();
while files.size(file)<=200000 do
    sleep(1000)
end;
audio.stop();
print 'Recorded',audio.len(),'ms in ',
    files.size(file),'bytes.';
→ Recorded 2900.158 ms in 255838 bytes.
→ 88.215193793 kB/s
```

## audio.play

- function play() → null
- function play(file) → null

*Permissions:* **Read(file)**

Without argument, starts or continues playing the currently open sound file.

With one argument, directly starts playing a sound file (.mp3, .wav, .au or such). The file name is relative to the current directory (see 3.1 (p. 41)). When

the sound file has finished playing, it is closed.

This function immediately returns, before playing completes. Exceptions can therefore be thrown anywhere in the following code. Use `audio.wait` (p. 70) to wait for completion.

Throws `ErrInUse` if the sound unit is busy playing or recording another sound.

Without argument, throws `ErrNotReady` if no file has been opened, and throws `ErrArgument` if the current playing position is outside the file.

```
audio.play("c:\\documents\\audio\\Hello.mp3")
```

## `audio.pos`

- `function pos() → Number`
- `function pos(ms) → Number`

Without arguments, returns the playing position in the current file, in milliseconds from the start.

With one argument, set the playing position to `ms` milliseconds.

Throws `ErrNotReady` if no file has been opened.

With one argument, throws `ErrInUse` if the file is being played or recorded.

```
// Open a file and play seconds 5 to 12
audio.open('sample.wav');
audio.pos(5000);
audio.play();
sleep(7000);
print audio.pos();
audio.stop()
→ 11850
```

## `audio.record`

- `function record(gain=100) → null`

Compatibility of function <code>audio.record</code>	
Sony Ericsson phones cannot record phone conversations	<code>ErrInUse</code>
Sony Ericsson phones do not reliably detect unsupported sample rates, resulting in mismatches between sampled and played rates.	

Record sound from the microphone or from an ongoing phone conversation (mixing microphone and incoming phone signal). `gain` is the recording gain, a number between 0 (minimum or automatic) and 100 (maximum). Default gain is 100. Setting the gain to a negative value sets it to 0, setting it to a value greater than 100 sets it to 100.

The audio data is appended to the current file. Use `audio.cut` (p. 65) to truncate the file and set the recording position.

This function immediately returns, before recording completes. Exceptions can therefore be thrown anywhere in the following code. Use `audio.stop` (p. 69) to stop recording.

Throws `ErrInUse` if a file is already being played or recorded. Throws `ErrNotSupported` if the file format does not support recording, or if the sample rate is not supported.

To add 20 seconds of recorded sound at the end of an existing audio file `sample.wav`:

```
audio.open('sample.wav', audio.rw);
audio.record();
sleep(20000);
audio.stop()
```

## `audio.stop`

- `function stop() → null`

Stops the currently playing sound, or the current recording.

## audio.volume

- function volume() → Number
- function volume(percent) → Number

Returns the current sound output volume and optionally changes it. The volume is a number between 0 (mute) and 100 (loudest). Default volume is 50. Setting the volume to a negative value sets it to 0, setting it to a value greater than 100 sets it to 100.

On most devices, changing the volume while a sound is playing has immediate effect.

```
audio.play("c:\\documents\\audio\\HomeBox.mp3");
while audio.busy() do
  sleep(100);
  audio.volume(audio.volume()-10) // fade out
end
```

## audio.wait

- function wait() → null

Waits until playing completes. Returns immediately if no sound is playing.

This function checks only the current `m` process: it will return immediately if the sound unit is in use by another process (inside or outside of `m`).

```
for i=1 to 10 do
  audio.wait(); audio.beep(440, 500);
  audio.wait(); audio.beep(330, 500)
end
```

## 3.5 Module contacts: Contacts Database

This module allows to read and manipulate the contacts stored on the phone. In the phone's database, a contact is identified by its id, an integer number.



## Contact Fields

In `m`, a contact is represented as an array whose elements are the fields of the contact. Fields are identified by their (array) keys. `m` recognizes the following keys, with the corresponding data type:

Key	Meaning
<code>adr</code>	Address (street)
<code>birth</code>	Birthday
<code>cell</code>	Cellphone number
<code>company</code>	Company name
<code>country</code>	Country
<code>email</code>	e-mail address
<code>extadr</code>	Additional address
<code>extname</code>	Additional name
<code>fax</code>	Fax number
<code>fname</code>	First name
<code>loc</code>	Locality (city)
<code>name</code>	(Family) name

Key	Meaning
<code>note</code>	Contact note
<code>pager</code>	Pager number
<code>phone</code>	Voice phone number
<code>pict</code>	Picture image data
<code>po</code>	Post Office
<code>region</code>	Region
<code>ring</code>	Ringtone file name
<code>text</code>	Free text
<code>title</code>	Job Title
<code>url</code>	Website URL
<code>video</code>	Video phone number
<code>zip</code>	Post Code

Key names are not case sensitive.

The order of fields in the array describing a contact is arbitrary. Arrays returned by functions in this module always start with the two fields `name` and `fname`, if these fields exist.

Address and phone number fields can have one of the following suffixes:

Suffix	Meaning
<code>.home</code>	Home address or phone
<code>.work</code>	Work address or phone

For instance, `phone.home` refers to the home phone number, `phone.work` to the work phone number. `phone` without suffix is unspecified.

Most fields are represented as strings. There are two exceptions:

- `birth`: The birthday is stored as a number indicating the seconds since year zero. This is the format used by module `time` (p. 131).
- `pict`: The picture is stored as an array containing the image data, typically in JPEG format. Example functions to load or store a the picture of a contact `c`:

```

use io
function loadpict(file, c)
  f=io.open(file);
  s=io.read(f, io.size(f)); // read whole file
  io.close(f);
  c["pict"]=code(s) // string to byte array
end
function storepict(c, file)
  if c["pict"]#null then
    s=char(c["pict"]); // byte array to string
    f=io.create(file);
    io.write(f, s);
    io.close(f)
  end
end
end

```

Note that the builtin contacts application in the phone may not support all keys, or display some of them in a strange way. Furthermore, not all applications clearly separate home from work data. Hence, the cell phone number of a person is sometimes stored as `cell`, sometimes as `cell.work` or as `cell.home`.

The functions of this module throw `ExcInvalidParam` if a contact array has no keys, or `ErrBadName` if a contact array has a key which is not in the above table.

## contacts.add

- function add(contact) → Number

*Permissions:* WriteApp

Add a contact to the database, and return its id. The contact must be an array with keys from the above tables.

```

c=["name": "Shakespeare",
  "fname": "William",
  "loc.home": "Stratford-upon-Avon"],
  "loc.work": "London",
  "birth": time.num("1564-04-23")];
contacts.add(c)
→ 114

```

## `contacts.delete`

- function `delete(id) → null`

*Permissions:* `WriteApp`

Delete the contact with the given `id`.

Throws `ErrNotFound` if there is no such contact.

```
// delete the contact added in the add example
contacts.delete(114)
```

## `contacts.find`

- function `find(text=null, keys=["name", "fname"], sort=[]) → Array`

*Permissions:* `ReadApp`

Searches the contact database for entries matching `text` considering the fields specified in `keys`, and returns the ids of the matching contacts sorted by the fields specified in `sort`:

- If `text=null`, all entries are returned, and `keys` is ignored.
- If `text#null`, searches the contact database for all entries matching the words in `text` when considering the fields defined by `keys`. Both `text` and all fields from the database are split into words (sequences of characters or digits) before comparing them. An entry matches if all of the words in `text` are found in any of the fields considered. Words can also be abbreviated: `William` matches both `W` or `Will` in the search text.  
If `keys` defines a single field, it can be a string, otherwise it must be an array of strings.
- If `sort=[]`, the ids are sorted by their ascending numeric value.
- If `sort` is a string, the ids are sorted by the corresponding field.
- If `sort` is an array, the ids are sorted by the corresponding fields, from highest to lowest sort order.

Throws `ErrArgument` if there are more than 32 keys or sort keys specified.

```
// get the number of contacts in the database
print len(contacts.find())
→ 104
// print these contacts, sorted by name and first name
for id in contacts.find(null,null,["name", "fname"]) do
  c=contacts.get(id);
  print c[1], c[0]
end
→ ...
  William Shakespeare

// Will matches William; so does W
print contacts.find("Will Shakespeare")
→ [114]
print contacts.find("W. Shakespeare")
→ [114]
// get the ids of everybody living or working in London
print contacts.find("London", "loc")
→ [45,67,89,90,91,114]
// Stratford-upon-Avon is considered three words,
// so Avon matches
print contacts.find("Avon", "loc")
→ [114]
```

## contacts.findnr

- function `findnr(number, digits=8) → Array`

*Permissions:* `ReadApp`

Retrieves the ids of the entries matching the given phone number. Only the last `digits` in `number` are considered when comparing. The minimum for `digits` is 7.

This function is much faster than `find`, and more useful, as it only looks at digits, and the end of the phone numbers.

Throws `ExcValueOutOfRange` if `digits` is out of range.

```
print contacts.findnr("+41(079)7654321", 9)
→ [28]
```

## `contacts.get`

- function `get(id, keys=null) → Array`

*Permissions:* `ReadApp`

Get fields of the contact with `id` `id`. If `keys=null`, returns all fields defined for the contact. If `keys#null`, returns only the fields specified in `keys`. `keys` can be a single string specifying a single field, or an array specifying multiple fields.

If they exist, the fields `name` and/or `fname` are at the beginning of the returned array.

Throws `ErrNotFound` if there is no contact with this `id`; throws `ErrArgument` if there are more than 32 `keys` specified.

```
c=contacts.get(114);
print c
→ [Shakespeare,William,Stratford-upon-Avon,London,
  49365849600]
print time.str(c["birth"])
→ 1564-04-23 00:00:00
print contacts.get(114, ["name", "fname"])
→ [Shakespeare,William]
c=contacts.get(114, "loc");
print c
→ [Stratford-upon-Avon,London]
print keys(c)
→ [loc.home,loc.work]
```

## `contacts.labels`

- function `labels(keys=null) → Array`

Get labels for the fields. Labels are language dependent. `keys` is interpreted as follows:

- If `keys=null`, returns all standard labels.
- If `keys` is a string, returns the label(s) for the corresponding field(s).
- If `keys` is an array, returns the labels for the corresponding fields.

Throws `ErrArgument` if there are more than 32 keys specified.

Suffices `(.home, .work)` can be used as keys, but not as field suffices: `labels()` throws `ErrBadName` in this case.

If they exist, the labels for `name` and/or `fname` are at the beginning of the returned array.

The label array has the same keys as a contact.

```
l=contacts.labels();
print l
→ [Last name,First name,Tel. (home),Mobile
   (home),Fax (home),E-mail (home),Web addr. (home),
   Street (home),...<46>]
l["title"]
→ Job title
// print a contact with all its labels
c=contacts.get(114);
for k in keys(c) do
  print l[k], "-", c[k]
end
→ Last name - Shakespeare
   First name - William
   City (home) - Stratford-upon-Avon
   City (business) - London
   Birthday - 49365849600
// get all work related labels
print contacts.labels([".work"])
→ [Tel. (business),Mobile (business),Fax
   (business),E-mail (business),Web addr. (bus.),Street
   (business),...<12>]
contacts.labels("phone.work")
→ ErrBadName thrown
```

## contacts.new

- function `new(time)` → Array

*Permissions:* `ReadApp`

Returns the list of contacts modified since the specified point in time. `time` is the number of seconds since year 0 UTC. See also module `time` (p. 131).

```
// get the entries changed within the last ten minutes
print contacts.new(time.utc()-10*60)
→ [114]
```

## `contacts.own`

- function `own()` → Number

*Permissions:* `ReadApp`

- function `own(id)` → Number

*Permissions:* `ReadApp+WriteApp`

There is a single contact in the database which can be marked as own contact, indicating the owner of the phone (or any other particular person). Some phones can use this information to quickly send a vCard<sup>4</sup> of the phone owner.

Without an argument, the id of this contact is returned. With an argument, the own contact id is set to `id`, and the old one is returned.

Returns `-1` if no own contact has been set, or it has been deleted.

Throws `ErrNotFound` if there is no contact with this id.

```
// if there is no owner, make it the first Shakespeare
if contacts.own()=-1 then
  ids=contacts.find("Shakespeare");
  if len(ids)>0 then
    contacts.own(ids[0])
  end
end
```

## `contacts.set`

- function `set(id, contact)` → null

*Permissions:* `WriteApp`

Updates the contact with id `id`, updating or adding fields in array `contact`. `contact` must be an array with keys from the above tables.

<sup>4</sup>A standard defined by the Internet Mail Consortium, see [www.imc.org/pdi/vcardoverview.html](http://www.imc.org/pdi/vcardoverview.html).

Fields already existing in the database are updated, the other fields are added. Fields not in the array are not modified. Fields which are `null` in the array are removed from the contact.

```
// Replace all +41 1 numbers by +41 44
const fields=["phone", "fax", "cell", "pager"]);
for id in contacts.find() do
  c=contacts.get(id, fields);
  m=false;
  for i=0 to len(c)-1 do
    // field could be null or too short
    if c[i]!=null then
      n=trim(c[i]);
      if len(n)>=11 then
        // replace +411 by +4144
        if substr(n,0,4)="+411" then
          c[i]="+4144" + substr(n, 4); m=true
        // replace +41 1 by +41 44
        elsif substr(n,0,5)="+41 1" then
          c[i]="+41 44" + substr(n, 5); m=true
        end
      end
    end
  end;
  if m then
    contacts.set(id, c)
  end
end
```

## 3.6 Module files: File and Directory Access

This module provides access to files and directories of the underlying operating system, including a function to send a file via different messaging interfaces (“send as”).

To read and write files, use module [io](#) (p. 111).

If not absolute, pathes are always relative to the current directory. See also section 3.1 (p. 41).



Some functions of this module allow the use of *file patterns*: these may contain the wildcards `*` matching any number of characters, and `'?'` matching a single character. For instance, the pattern `d:\documents\mShell\*Test.*` matches any file in directory `\documents\mShell` on drive `D`: whose name ends with `Test`.

Many of the functions in this module can render a mobile phone completely unusable, e.g. by deleting system configuration data, or by overwriting sensitive files. Make sure you regularly back up your mobile phone, and inform yourself how to reset your phone to factory status. You have been warned!



## `files.attr`

- function `attr(path) → Number`

*Permissions:* `Read(path)`

- function `attr(path, newattr) → Number`

*Permissions:* `Read+Write(path)`

Gets or sets the attribute bits of a file. With one argument, returns the attribute bits of the file or directory `path`. With two arguments, returns the old file attributes, and sets the new attributes of `path`.

The attribute bits define the characteristics of a file:

- `const arch = 32` File or directory has the archive bit set.
- `const dir = 16` Path references a directory.
- `const hidden = 2` File or directory is hidden (invisible).
- `const ro = 1` File or directory is read-only.
- `const sys = 4` File or directory has the system bit set.
- `const all = 55` All attribute bits set.

The status of the `files.dir` attribute cannot be changed.

Use the bitwise or operator `|` to combine single bits; use the bitwise and operator `&` to check for single bits.

```
// make the file "secret.dat" read-only and invisible
files.attr("secret.dat", files.ro | files.hidden);
// check whether a path is a directory
print
    files.attr("c:\\documents\\mShell") & files.dir # 0
→ true
```

See also: [files.scan](#) (p. 84)

## files.copy

- function copy(srcpattern, destdir, recursive=false) → Number  
/r:recursive

*Permissions:* Read(srcpattern)+Write(destdir)

Copies a file or all files matching `srcpattern` to another directory `destdir`. If `recursive=true`, or `/r` is specified in interactive mode, also copies all files matching the file part of `srcpattern` in all subdirectories of the directory part of `srcpattern`, and creates the corresponding subdirectories in `destdir`.

Returns the number of files copied.

In interactive shells, this function is available as `cp`.

```
print files.copy("secret.dat", "d:\\")
→ 1
// copy all m scripts from drive C: to drive D:
files.copy("c:\\documents\\mShell\\*.m",
           "d:\\documents\\mShell", true)

m>cp c:\documents\mShell\*.m d:\documents\mShell/r
```

The last two statements (the second in interactive mode) are equivalent.

## files.delete

- function delete(pattern, recursive=false) → Number  
/r:recursive

*Permissions:* Write(pattern)

Deletes a file or all files matching `pattern`. If `recursive=true`, or `/r` is specified in interactive mode, also deletes all files matching the file part of `pattern` in all subdirectories of the directory part of `pattern`.

Returns the number of files deleted.

In interactive shells, this function is available as `del`.

```
print files.delete("secret.dat");  
→ 1  
// delete all m scripts from drive C:  
files.delete("c:\\documents\\mShell\\*.m", true)  
  
m>del c:\\documents\\mShell\\*.m/r
```

The last two statements (the second in interactive mode) are equivalent.

See also: `files.rmdir` (p. 83)

## `files.edit`

- function `edit(path, cursor=0) → null`

*Permissions:* `Read+Write(path)`

Loads the file `path` into the builtin editor, and shows the editor. Any previously loaded file (e.g. a script being edited) will be saved first. The cursor is moved to position `cursor` in the file. The character encoding applied is determined by the `encoding` property (see A.3 (p. 161)).

In interactive shells, this function is available as `edit`.

```
// edit an XML file  
files.edit("\\documents\\MMS\\Sample.xml")
```

## `files.exists`

- function `exists(path) → Boolean`

*Permissions:* `Read(path)`

Returns `true` if the file or directory denoted by `path` exists, `false` if there is no such file or directory.

```
print files.exists("c:\\documents\\mShell")
→ true
```

## files.mkdir

- function mkdir(path, all=false) → null  
/a:all

*Permissions:* Write(path)

Create a new directory `path`. `path` can be relative to the current directory, or absolute. See also section 3.1 (p. 41).

If `all=false`, `mkdir` creates just one directory. If `all=true`, or `/a` is specified in interactive mode, all directories down to the last in `path` are created, as necessary.

In interactive shells, this function is available as `md`.

```
mkdir("subdir");
mkdir("../otherdir");
mkdir("c:\\documents\\mShell", true)

m>md c:\\documents\\mShell/a
```

The last two statements (the second in interactive mode) are equivalent.

## files.move

- function move(srcpattern, destpath, recursive=false) → Number  
/r:recursive

*Permissions:* Read+Write(srcpattern), Write(destdir)

Moves a file or all files matching `srcpattern` to another directory `destdir`. If `recursive=true`, or `/r` is specified in interactive mode, also moves all files matching the file part of `srcpattern` in all subdirectories of the directory part of `srcpattern`, removes and creates the corresponding subdirectories in `destdir`.

Returns the number of files moved.

In interactive shells, this function is available as `mv`.

```
print files.move("secret.dat", "d:\\")
→ 1
// move all m scripts from drive C: to drive D:
files.move("c:\\documents\\mShell\\*.m",
           "d:\\documents\\mShell", true)

m>mv c:\\documents\\mShell\\*.m d:\\documents\\mShell/r
```

The last two statements (the second in interactive mode) are equivalent.

## files.rename

- function rename(oldfile, newfile) → null

*Permissions:* Write(oldfile)+Write(newfile)

Renames the file or directory oldfile to newfile. This function does not support wildcards.

```
files.rename("secret.dat", "topsecret.dat")
```

## files.rmdir

- function rmdir(path, recursive=false) → Number  
/r:recursive

*Permissions:* Write(path)

Removes the directory path. If recursive=false, the directory must be empty before it can be removed. If recursive=true, or /r is specified in interactive mode, the directory with all its contents and subdirectories will be removed.

Returns the number of directories and files removed.

In interactive shells, this function is available as rd.

```
print rmdir("subdir")
→ 1
rmdir("../otherdir");
rmdir("c:\\myfiles\\images", true)

m>rd c:\\myfiles\\images/r
→ (number of items removed)
```

The last two statements (the second in interactive mode) are equivalent: they both remove the directory `images` with all its contents.

## files.roots

- function `roots()` → Array

Returns an array with all accessible file system roots (drives).

```
print files.roots()
→ [A:,C:,D:,Z:]
```

## files.scan

- function `scan(pattern, attr=0, mask=files.dir | files.hidden | files.sys)` → Array

*Permissions:* `Read(pattern)`

Returns an array with all directory entries whose name matches `pattern` and whose attribute bits defined by `mask` match `attr`: a file path matches if `files.attr(path) & mask = attr & mask`.

Example values for `attr` and `mask`:

- The default values exclude directories, hidden and system files.
- `attr=files.dir` returns only directories.
- `mask=0` ignores all attributes and thus returns all entries.
- `attr=files.ro` and `mask=files.ro` return only read only files and directories.

- `attr=files.arch` and `mask=files.dir|files.arch` return only files with the archive bit set.

The file names returned do not contain the directory part defined by `pattern`, and are sorted by name.

```
// search the application directory for DLL files
print files.scan(system.appdir+"*.dll")
→ [Array_mm.dll,Audio_mm.dll,...]
// search the document directory for hidden files only
print files.scan(system.docdir+"*",files.hidden)
→ [10204299.act]
```

## files.send

- function `send(path, subject=null) → null`

*Permissions:* `Read(path)`

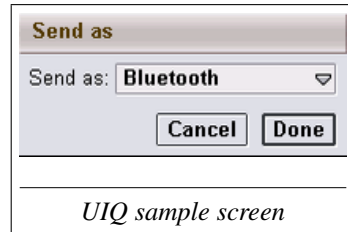
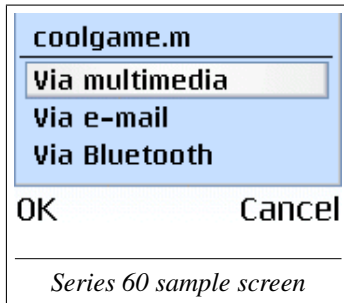
Compatibility of function <code>files.send</code>	
Nokia phones before Symbian 8	Call is ignored

Sends the file `path` over a messaging channel chosen by the user (“Send as”). Channels typically include Bluetooth, MMS, and e-mail. The recipient and other channel dependent message details will be specified interactively.

`subject` is the subject of the message (if applicable). If `subject=null`, it defaults to `path` without the directory component.

In interactive shells, this function is available as `send`.

```
// send a script file
files.send(system.docdir+"coolgame.m",
           "The cool game I promised")
```



## files.size

- function `size(path) → Number`

*Permissions:* `Read(path)`

Returns the size in bytes of the file denoted by `path`. Returns 0 if `path` denotes a directory.

```
print files.size(system.appdir+"Audio_mm.dll")
→ 2956
```

## files.time

- function `time(path) → Number`

*Permissions:* `Read(path)`

- function `time(path, newtime) → Number`

*Permissions:* `Read+Write(path)`

Gets or sets the time when the file or directory denoted by `path` has been created or modified. The time is in seconds since midnight on January 1st of year 0. With one argument, returns the modification time of the file or directory `path`. With two arguments, returns the old modification time, and sets the new time.

```
print files.time("c:\\documents\\mShell")
→ 63276033444
```

See also module [time](#) (p. 131).



## 3.7 Module `graph`: Screen Graphics

This module supports drawing of arbitrary two-dimensional graphic objects and images from files on the screen. The module has its own view, which can be shown or hidden under programmatic control. When shown, it appears on top of the normal console window and hides it.

The view supports two modes: “console mode”, with the view covering the area of the `m` console, and “full screen mode”, with the view covering the entire screen. The default mode is “console”, but it can be changed any time by `graph.full` (p. 94).

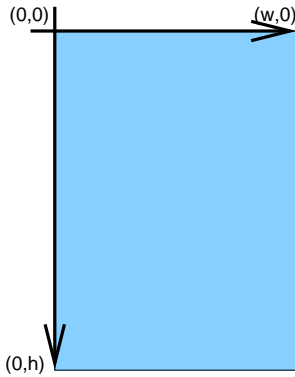
By default, the drawing area’s size (“canvas” size) corresponds to the console’s size, but it can be changed to any size which fits into memory via the `graph.size` (p. 106) function. If the canvas is bigger than the view, the origin of the view on the canvas can be specified via the `graph.show` (p. 105) function.

Graphic objects are added by calling the corresponding functions, and are drawn in the order they have been added: objects added later are drawn over objects added earlier. Objects are not drawn until `graph.show` (p. 105) is called, or the operating system requests redrawing.

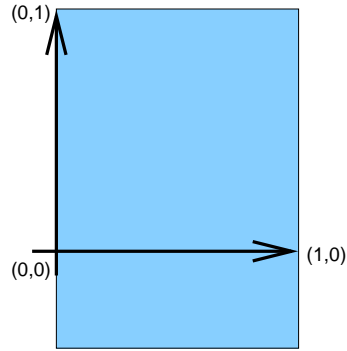
### Coordinates

Position and size of graphic objects are given by coordinates. This module supports two modes for specifying coordinates (see also `graph.scale` (p. 104)):

- *Unscaled*, with the unit being a single screen pixel, defining the area to draw on as a rectangle of integer width and height. Following conventions for pixel coordinates,  $y=0$  is at the top of the rectangle, and  $y$  increases downwards.
- *Scaled*, normalizing the rectangle to draw on as a square with sides of length 1, and an additional rectangle on the right for  $x>1$  (typically on Series 60 devices), or at the bottom for  $y<0$  (typically on UIQ devices). Following conventions for mathematical coordinates,  $y=0$  is at the bottom of the square, and  $y$  increases upwards.



Unscaled (pixels)



Scaled (unit square)

## Colors

Colors for the graphic are expressed as RGB, i.e. as the three intensities of red, green and blue. In `m`, there are two ways to specify an RGB value:

- As an array of three color intensities between 0 and 1. For instance, `[0.5, 0, 0.5]` specifies a dark magenta (50% red and 50% blue).
- As an integer encoding the three color intensities between 0 and 255 as `red shl 16 | green shl 8 | blue`. This is typically written in hexadecimal notation as `0xrrggbb`. For instance, `0x800080` is (after rounding) equivalent to `[0.5, 0, 0.5]`.

Eight standard colors are predefined as module constants:

- `const black = 0x000000`
- `const white = 0xffffffff`
- `const red = 0xff0000`
- `const green = 0x00ff00`
- `const blue = 0x0000ff`
- `const yellow = 0xffff00`
- `const cyan = 0x00ffff`
- `const magenta = 0xff00ff`

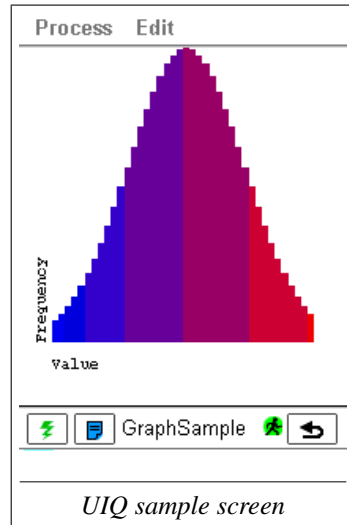
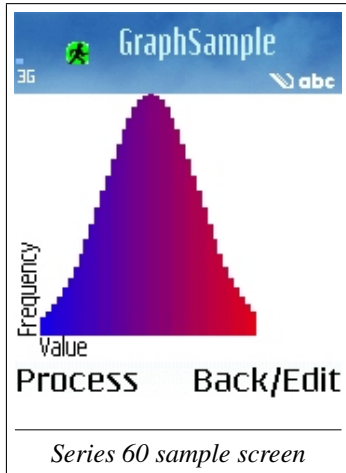
The view itself has a background color (set via `graph.bg` (p. 90)), which initially is white. Graphic items drawn on the background generally have two colors:

- The *pen color* defines the color in which lines, texts and outlines are drawn. It can also be set to `false`, so no outlines are drawn. It is initially black, and set via `graph.pen` (p. 100).
- The *brush color* defines the color by which areas are filled. It can also be set to `false`, so areas are not filled. It is initially `false`, and set via `graph.brush` (p. 90).

## Simple Example

The following example draws the graph of a normal distribution around the average 0.5, coloring it from almost pure blue to almost pure red:

```
// use the normalized 0 to 1 coordinate system
graph.scale(true);
h=0.02;
for x=0.1 to 0.9 by h do
    t=-4*(x-0.5); y=math.exp(-t*t)*0.9;
    color=[x,0,1-x];
    graph.pen(color); graph.brush(color);
    graph.rect(x,0.1,h,y)
end;
graph.pen(graph.black);
graph.text(0.1, h, "Value");
graph.text(0.1-h, 0.1, "Frequency", graph.up);
graph.show();
```



## graph.bg

- function bg(color) → Array
- function bg() → Array

Gets or sets the background color of the graph view. With one argument, sets the background color, and returns the old background color, as an array of red, green and blue intensities. Without arguments, returns the current background color.

See section 3.7 (p. 88) for the definition of colors.

```
// set the background color to a light gray
graph.bg([0.9,0.9,0.9])
```

## graph.brush

- function brush(color) → Array
- function brush() → Array

Gets or sets the brush color. This is the color used to fill areas surrounded

by objects. With one argument, sets the brush color or disables it (if `color=false`), and returns the old brush color as an array of red, green and blue intensities, or `false` if the brush was disabled. Subsequently added objects will use the new brush color.

Without arguments, returns the current brush color.

By default, the brush is disabled. See section 3.7 (p. 88) for the definition of colors.

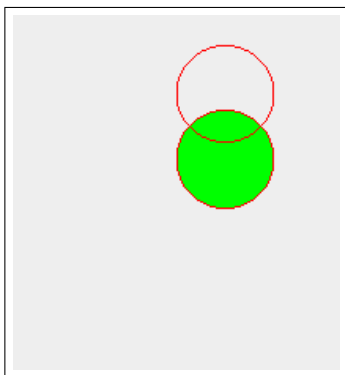
```
// fill the objects with white
graph.brush(graph.white)
```

### `graph.circle`

- function `circle(x, y, diameter) → null`

Draws a circle in the square defined by the corners `(x,y)` and `(x+diameter,y+diameter)`. The outline is drawn with the current pen color, and the circle is filled with the current brush color.

```
graph.scale(true);
graph.pen(graph.red);
graph.brush(graph.green); // fill with green
graph.circle(0.5, 0.4, 0.3);
graph.brush(false); // do not fill
graph.circle(0.5, 0.6, 0.3);
```



*Sample m screen*

## graph.clear

- `function clear() → null`

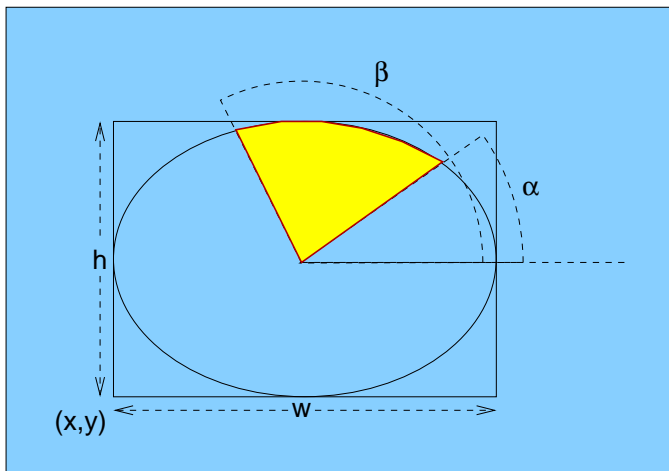
Removes all objects from the view, so only an empty background is drawn.

## graph.ellipse

- `function ellipse(x, y, w, h) → null`
- `function ellipse(x, y, w, h, alpha, beta) → null`

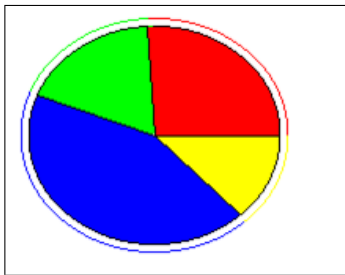
Draws an ellipse, an arc or a pie slice:

- With four arguments, draws an ellipse into the rectangle with corner at  $x, y$ , width  $w$  and height  $h$ . The outline is drawn with the current pen color, and the ellipse is filled with the current brush color.
- With six arguments and the brush enabled, draws the outline of a pie slice with the current pen color, and fills it with the current brush color. The pie is defined by two angles  $\alpha$  and  $\beta$  measured in degrees from the x axis, on a circle around the center of the ellipse:



- With six arguments and the brush disabled, draws just the arc, i.e. the part of the pie on the outline of the ellipse.

```
// draw an elliptic pie, with parallel arcs
percent=[26, 18, 43, 13];
colors=[graph.red,graph.green,graph.blue,graph.yellow];
alpha=0;
for i=0 to len(percent)-1 do
    beta=alpha+360*percent[i]/100;
    // the pie slice (brush enabled)
    graph.pen(graph.black); graph.brush(colors[i]);
    graph.ellipse(10, 10, 160, 140, alpha, beta);
    // the parallel arc (brush disabled)
    graph.pen(colors[i]); graph.brush(false);
    graph.ellipse(5, 5, 170, 150, alpha, beta);
    alpha=beta
end;
graph.show()
```



*Sample m screen*

## graph.font

- function font(font) → Array
- function font() → Array

Gets or sets the text font. With one argument, sets the current font, and returns the old font. Subsequently via [graph.text](#) (p. 107) added texts will use the new font. Without arguments, returns the current font.

The default font is the m console font. See [ui.mfont](#) (p. 144) for the definition of fonts, and [graph.text](#) (p. 107) for an example using fonts.

## graph.full

- function full() → Array
- function full(enabled) → Array

Compatibility of function graph.full	
Sony Ericsson phones <sup>a</sup> .	Restricted menu access

<sup>a</sup>In full screen mode, menus can only be accessed with the jog dial. Once activated, the menu bar will stay on top of the view until `graph.show` is called again.

Without arguments, returns the size of the view in the current mode, scaled if in scaled mode.

With one argument, enables (`enabled=true`) or disables (`enabled=false`) full screen mode, and returns the new view size, scaled if in scaled mode. Note that this does *not* change the size of the canvas; the canvas size can only be changed with `graph.size` (p. 106).

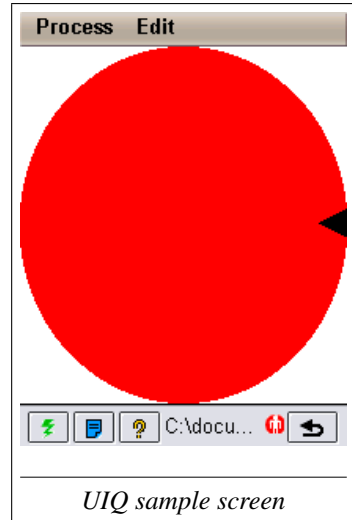
The following function fills the screen (not the canvas) with an ellipse in a given color:

```
function fill(color)
  graph.clear();
  graph.pen(color); graph.brush(color);
  s=graph.full(); // get screen size
  graph.ellipse(0, 0, s[0], s[1])
end
```



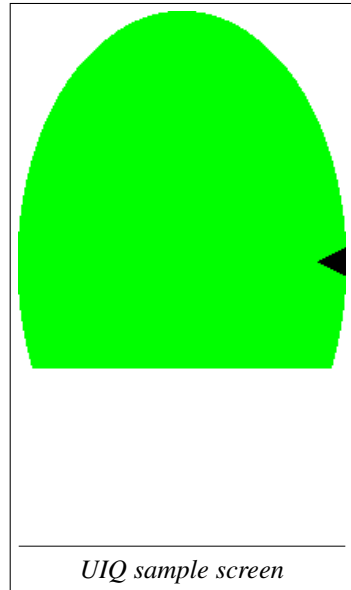
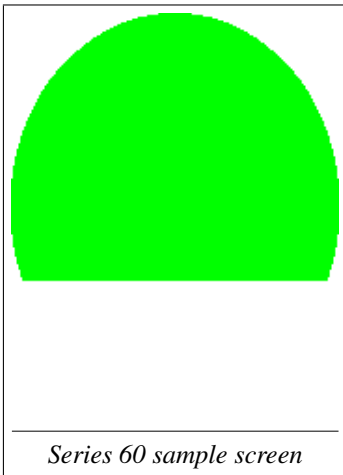
Drawing a red ellipse in console mode just fills the console view, as usual:

```
graph.full(false);  
fill(graph.red);  
graph.show();
```



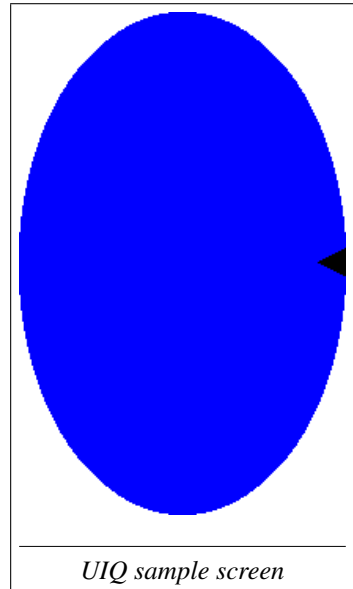
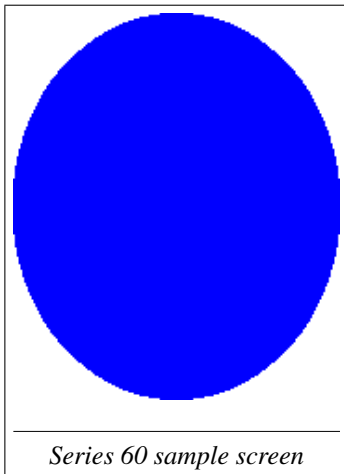
Drawing a green ellipse after changing to full screen mode truncates the ellipse to the console view size (assuming the canvas size wasn't changed):

```
graph.full(true);  
fill(graph.green);  
graph.show()
```



Drawing a blue ellipse after setting the canvas size to the view size fills the entire screen with the ellipse:

```
s=graph.full(true);  
graph.size(s[0], s[1]);  
fill(graph.blue);  
graph.show()
```



### `graph.get`

- function `get(x, y) → Number`
- function `get(x, y, w) → Array`
- function `get(x, y, w, h) → Array`

Gets a pixel, a scan line or a rectangle from the current image.

With two arguments, returns the color of the pixel at  $(x, y)$  as a single integer (see section 3.7 (p. 88)).

With three arguments, returns an array with the pixel colors of the horizontal

line of length  $w$  starting at  $(x, y)$ .

With four arguments, returns a matrix with the pixel colors of the rectangle with corner  $(x, y)$ , width  $w$  and height  $h$ .

In scaled mode, coordinates and dimensions are scaled.

See also [graph.put](#) (p. 101).

## graph.hide

- function `hide()` → null

Hides the graph view, showing the standard process view, or any previous view. If the graph view is not shown, this call is ignored.

## graph.icon

- function `icon(path, transparent=null)` → Native Object

*Permissions:* `Read(path)`

- function `icon(data, transparent=null)` → Native Object
- function `icon(data, maskData)` → Native Object
- function `icon(icon)` → Native Object

Creates an icon from an image file, or from color data, and returns the icon object. Icons may have an optional transparency mask, defining which pixels are opaque (drawn) and which are transparent (not drawn) when drawing the icon with [graph.put](#) (p. 101).

With a single `path` argument, loads an image from the file at `path`, and returns it as an icon. The image file formats supported vary from device to device, but usually include BMP, GIF, JPEG and PNG formats. If the image has transparency information, it is also loaded to define the icon's transparency mask. Alternatively, if `transparent` is a number, all pixels of this color are assumed transparent.

With a single `data` argument, the icon's image is defined by the colors in `data`. `data` is typically a matrix as returned by [graph.get](#) (p. 97), but can also be a single pixel or a scan line. If `transparent` is a number, all pixels of this color are assumed transparent. Alternatively, the matrix `maskData` can define transparency on a pixel by pixel basis: all black (zero) pixels in

`maskData` are assumed transparent. `maskData` must have the same dimensions as `data`.

With a single `icon` argument, a copy of the icon is created and returned, e.g. to scale it while still keeping the original.

Use `graph.size` (p. 106) to obtain the size of an icon, or to rescale it.

Large icons, e.g. those produced by a high resolution camera, consume considerable memory.

```
// load the icon
i=graph.icon("mShell.png")
// get its size
graph.size(i)
→ [156,92]
// draw it
graph.put(20,20,i)
// copy the icon
i2=graph.icon(i);
// scale the copy into a 80x80 square and draw it
graph.size(i2,80,80);
graph.put(20,120,i2);
graph.show()
```



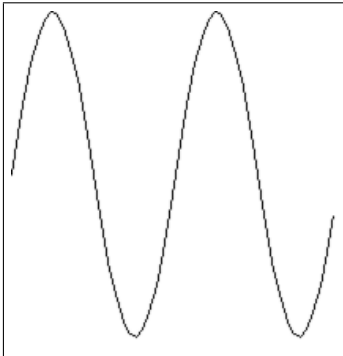
*Sample m screen*

## `graph.line`

- function `line(x1, y1, x2, y2) → null`

Draws a line from  $(x1, y1)$  to  $(x2, y2)$ , using the current pen color.

```
// plot a sine wave from 0 to 4 pi
graph.scale(true);
x1=0; y1=0;
for x=0 to 1 by 0.02 do
  y=(math.sin(4*math.pi*x)+1)/2;
  if x>0 then graph.line(x1,y1,x,y) end;
  x1=x; y1=y
end;
graph.show()
```



*Sample m screen*

## graph.pen

- function pen(color) → Array
- function pen() → Array

Gets or sets the pen color. This is the color used to draw the outlines of objects. With one argument, sets the pen color or disables it (if `color=false`), and returns the old pen color as an array of red, green and blue intensities, or `false` if the pen was disabled. Subsequently added objects will use the new pen color.

Without arguments, returns the current pen color.

The default pen color is black. See section 3.7 (p. 88) for the definition of colors.

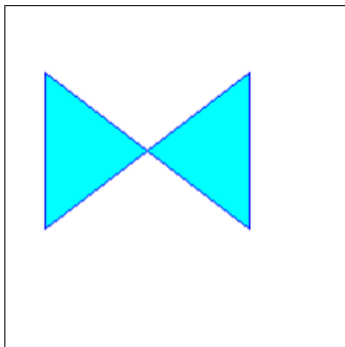
```
// use a slightly dark magenta pen
graph.pen(0xc000c0)
```

## `graph.poly`

- `function poly(x, y) → null`

Draws a closed polygon following the points given by `x` and `y`. `x` and `y` must be two arrays of identical length. The polygon's edges are lines from  $(x[i], y[i])$  to  $(x[i+1], y[i+1])$  ( $0 \leq i < \text{len}(x) - 1$ ), with the last (closing) line going from  $(x[\text{len}(x)-1], y[\text{len}(x)-1])$  to  $(x[0], y[0])$ . The lines of the polygon are drawn with current pen color, and the polygon's interior (or interiors) are filled with the current brush color.

```
// draw a blue bowtie, filled with cyan
graph.pen(graph.blue); graph.brush(graph.cyan);
graph.poly([20,150,150,20],[40,140,40,140]);
graph.show()
```



*Sample m screen*

## `graph.put`

- `function put(x, y, color) → null`
- `function put(x, y, icon) → null`

Draws a single pixel, a scan line or a rectangle, or draws an icon.

If `color` is a number, sets the pixel at  $(x, y)$  to the color `color`.

If `color` is an array of numbers, sets the pixels from  $(x, y)$  to  $(x+\text{len}(\text{color})-1, y)$  to the colors in `color`.

If `color` is a matrix of numbers, sets the rectangle with upper left corner  $(x, y)$ , height `len(data)` and width `len(data[0])` to the colors in `color`.

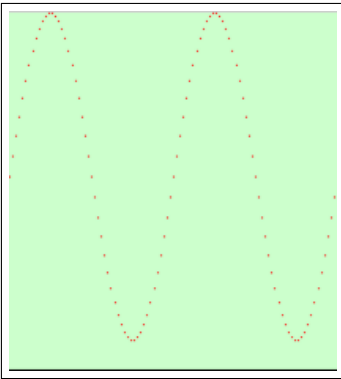
If the third parameter is an icon, draws `icon` with upper left corner  $(x, y)$ . If the icon has a mask, only opaque pixels are drawn.

In scaled mode,  $(x, y)$  are scaled, but always define the upper left corner of the rectangle.

Current pen and brush color do not affect what is being drawn.

A `graph.put` example drawing single points:

```
// plot a sine wave with single red points
graph.bg([0.8,1,0.8]); graph.clear();
graph.scale(true);
for x=0 to 1 by 0.01 do
  y=(math.sin(4*math.pi*x)+1)/2;
  graph.put(x,y,graph.red)
end;
graph.show()
```



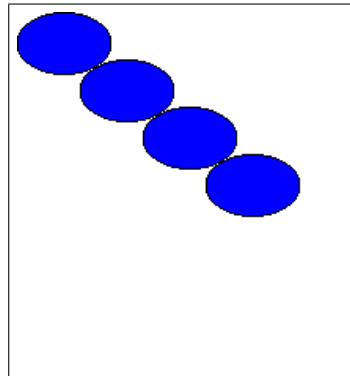
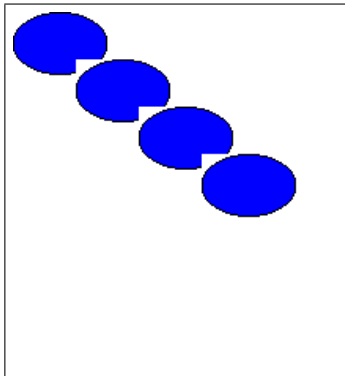
*Sample m screen*

A `graph.put` example drawing icons, with and without transparent back-



ground:

```
// Draw a blue ellipse
graph.brush(graph.blue);
graph.ellipse(0,0,60,40)
// Copy the ellipse and replicate it
data=graph.get(0,0,60,40);
for i=0 to 3 do
    graph.put(40*i,30*i,data)
end;
// The entire rectangle is overwritten
graph.show()
// Create an icon, making white transparent
icon=graph.icon(data, graph.white);
graph.clear()
// Replicate the icon
for i=0 to 3 do
    graph.put(40*i,30*i,icon)
end
// Only non-white pixels are overwritten
graph.show()
```



### graph.rect

- function rect(x, y, w, h) → null

Draws a rectangle between the corners (x,y) and (x+w,y+h). The outline

is drawn with the current pen color, and the rectangle is filled with the current brush color.

`rect(x, y, w, h)` produces the same as  
`poly([x, x+w, x+w, x], [y, y, y+h, y+h]).`

## graph.save

- function `save(path) → null`

*Permissions:* `Write(path)`

- function `save(path, x, y, w, h) → null`

*Permissions:* `Write(path)`

Saves the image produced by drawing to the file given by `path`. With one argument, saves the whole image. With five arguments, saves only the rectangle between the corners `(x, y)` and `(x+w, y+h)`.

The desired image file format is determined from the image file suffix. Supported file suffixes are `.gif` (GIF format), `.jpg` (JPEG format) and `.png` (PNG format).

Compatibility of saving to PNG	
Sony Ericsson phones	ErrNotSupported

```
// save the entire drawing to rates.jpg
graph.save("rates.jpg");
// save only the upper right quadrant to d:\rates.gif
graph.scale(true);
graph.save("d:\rates.gif", 0.5, 0.5, 0.5, 0.5)
```

## graph.scale

- function `scale(scaled) → Boolean`
- function `scale() → Boolean`

Gets or sets the current scaling mode. With one argument, sets the scaling mode to `scaled`, and returns the old scaling mode. Without an argument, returns the current scaling mode.

For information about scaling, see section 3.7 (p. 87).

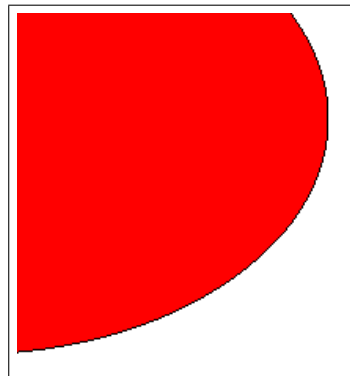
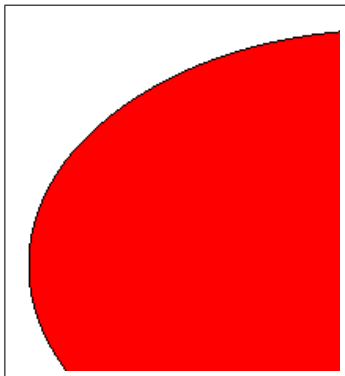
## `graph.show`

- function `show()` → null
- function `show(x, y)` → null

Shows the graph view, hiding the standard process view, and draws all objects added so far. If the graph view is already shown, it is redrawn.

With two arguments, also aligns the origin of the graph view with point  $(x, y)$  of the canvas. In unscaled mode, the origin of the view is in its upper left corner, and `graph.show(0,0)` aligns the upper left corner of the canvas with it. In scaled mode,  $x$  and  $y$  are scaled, and `graph.show(0,0)` aligns the lower left corner of the view with the lower left corner of the canvas.

```
// get the original size and create a canvas of 480x320
s=graph.size(480, 320)
// draw a red circle on it
graph.brush(graph.red);
graph.ellipse(10, 10, 460, 300)
// show its upper left quadrant
graph.show(0, 0)
// show its lower right quadrant
graph.show(480-s[0], 320-s[1])
```



## graph.size

- `function size() → Array`
- `function size(icon) → Array`
- `function size(text) → Array`
- `function size(w, h) → Array`
- `function size(icon, scale) → Array`
- `function size(icon, w, h) → Array`

Without arguments, returns the size (width and height) of the drawable area. The drawable area includes all the points in the rectangle between (0,0) and (`graph.size()[0]`, `graph.size()[1]`). In unscaled mode, `graph.size()` returns the width and height as number of pixels. In scaled mode, one of width or height will always be one.

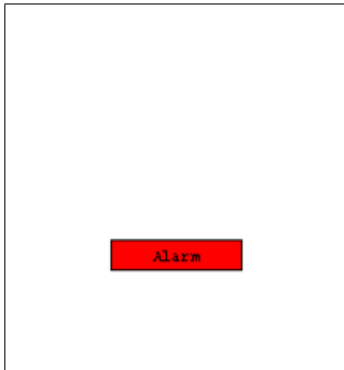
With one argument, returns the size (width and height) of the icon `icon`, or of `text` if it were drawn using the current font.

With two numeric arguments `w` and `h`, sets the size of the canvas to width `w` and height `h`, and returns the size of the old canvas (initially, the size of the canvas matches the size of the view). In unscaled mode, `w` and `h` are measured in pixels. In scaled mode, `w` and `h` are resizing factors (relative to the current size), and the scale is recalculated. See [graph.show](#) (p. 105) for an example using a canvas larger than the view.

With two arguments `icon` and a `scale`, scales the icon `icon` by the factor `scale`. Returns the old size (width and height) of the icon.

With three arguments, scales the icon `icon` to the width `w` and height `h`. Returns the old size (width and height) of the icon.

```
// get unscaled and scaled sizes
graph.scale(false);
print graph.size()
→ [208,227]
graph.scale(true);
print graph.size()
→ [1,1.0913461538]
// draw text centered in a red rectangle
text="Alarm"; x=0.5; y=0.2; w=0.6; h=0.2;
graph.brush(red); graph.rect(x, y, w, h);
s=graph.size(text);
graph.text(x+(w-s[0])/2,y+(h-s[1])/2,text);
graph.show()
```



*Sample m screen*

## graph.text

- function text(x, y, text, direction=0) → null

Draws text starting (the baseline of the first character) at point (x, y) using the current font. Text can be drawn horizontally or vertically:

- If direction=0, text is drawn horizontally.
- If direction>0, text is drawn vertically going up.
- If direction<0, text is drawn vertically going down.

Two indicate the direction, two constants are defined:

- `const up = 1` For vertical text going upwards.
- `const down = -1` For vertical text going downwards.

```
graph.pen(0x800080);
graph.text(50,70,"mShell");
graph.text(50,70,"mShell",graph.up);
old=graph.font(["SwissA", 24, true, false]);
graph.pen(0x808000);
graph.text(50,90,"mShell");
graph.text(50,90,"mShell",graph.down);
graph.font(old);
graph.show()
```



*Sample m screen*

## 3.8 Module `gsm`: GSM information

This module provides access to GSM (Global System for Mobile communication) related information. This includes identifiers and network information.

Please note that not all functions of this module are supported on all devices. Some functions may throw `ErrNotSupported`.

## `gsm.cid`

- function `cid()` → Number

*Permissions:* `ReadApp`

Gets the current CID (Cell Identity). Roughly speaking, a cell identifies the location of the phone: in a simplified view, each GSM cell corresponds to an antenna the phone is communicating with<sup>5</sup>. In cities, cells identify the location of the phone with a precision of a few hundred meters or even less. In remote locations, in particular on mountains, the distance to the antenna can be ten or more kilometers.

In practice, a specific location (e.g. an office) is typically covered by more than one cell, so the CID may change even if the phone doesn't move.

According to GSM specs, the CID is a number between 0 and 65535.

```
print gsm.cid()
→ 17437
```

## `gsm.net`

- function `net()` → Array

*Permissions:* `ReadApp`

Gets the current network as an array with the following keys:

Key	Contents
<code>mcc</code>	Mobile Country Code (MCC)
<code>mnc</code>	Mobile Network Code (MNC)
<code>short</code>	Short Network Name
<code>long</code>	Long Network Name
<code>lac</code>	Location Area Code (LAC)

To identify the current provider, MCC and MNC should be used. MCC and MNC of the home network are identical to the first three and two digits of the IMSI (see `gsm.imsi` (p. 111)).

Short and long name come from a database stored in the phone, so they may differ between phones for the same network.

---

<sup>5</sup>Usually, a single BTS (base transceiver station) covers multiple cells via sectorial antennas mounted on a single antenna tower.

```
n=gsm.net();  
print n  
→ 228,1,Swisscom,Swisscom,1616]  
print 100*n["mcc"]+n["mnc"]  
→ 22801  
print substr(gsm.imsi,0,5)  
→ 22801
```

## **gsm.new**

- function new(timeout=-1) → Boolean

*Permissions:* ReadApp

Waits until the current location information (typically the cell) changes, or until timeout milliseconds passed, if timeout >= 0.

Returns true if the location information changed, or false if the timeout expired.

The following code fragment waits ten seconds for a change in the location information, and prints the new cell if it changed.

```
if gsm.new(10000) then  
  print "In cell",gsm.cid()  
end
```

## **gsm.signal**

- function signal() → Number

*Permissions:* ReadApp

Gets the strength of the signal in the current network. The meaning of the returned value is device dependent. It may be a number between 0 (no signal) and 100 (strongest), or it may correspond to the number of signal strength bars normally shown on the display.

```
print gsm.signal()  
→ 89
```



## **gsm Constants**

- `const imei = phone identifier`

This constant contains the IMEI (International Mobile Equipment Identity) for the device `m` is running on. The IMEI is a fifteen digit unique identifier assigned to each device (cellphone). This number can also be queried directly by dialing `*#06#` on the phone.

```
print gsm.imei
→ 355023001234567
```

- `const imsi = subscriber identifier`

Compatibility of constant <code>imsi</code>	
Nokia 6600: the IMSI cannot be obtained.	<code>imsi=0000000000000000</code>

This constant contains the the IMSI (International Mobile Subscriber Identity) for the SIM card of the device `m` is running on. The IMSI is an up to fifteen digit unique identifier assigned to each subscriber (SIM card).

```
print gsm.imsi
→ 228011234567890
```

- `const number = own phone number`

Contains the own phone number, usually with country prefix.

```
print gsm.number
→ +41791234567
```

## **3.9 Module `io`: File and Stream Input/Output**

This module provides functions to read and write files or communication streams via the underlying operating system.

Some of the functions in this module can render a mobile phone completely unusable, e.g. by overwriting sensitive files. Make sure you regularly back up your mobile phone, and inform yourself how to reset your phone to factory



status. You have been warned!

Before file operations can be performed, a file has to be opened for reading or reading and writing. Opening a file returns a stream object which identifies the file for subsequent operations. When file operations are completed, the file should be closed<sup>6</sup>.

```
// open the standard autoexec.m script
f=io.open(system.appdir + "autoexec.m");
// read the first 28 bytes (characters)
s=io.read(f, 28);
print s;
→ /*
    Default autoexec script
// close the file
io.close(f)
```

There are two special files:

- `const stdin = standard input` Reads from the console.
- `const stdout = standard output` Writes to the console<sup>7</sup>

A file always has a *character encoding scheme* (CES) it uses when reading or writing UNICODE<sup>®</sup> characters. The following encoding schemes exist:

- `const raw = 0`

Only the low byte of each character is read or written, the high byte is assumed zero. The number of bytes written corresponds exactly to the number of characters. This is a good CES for reading and writing Latin characters, and the default CES.

- `const utf8 = 1`

Characters are encoded using UTF-8. This is a compact variable length encoding properly encoding all characters, but the number of bytes written is not easily predictable. Reading with the UTF-8 CES throws `ExcInvalidUTF8` if a character sequence not conforming to the UTF-8 standard is encountered.

- `const utf16le = 2`

Characters are encoded using UTF-16 LE (little endian, low byte first). Each character is read or written as two bytes, the number of bytes written is there-

<sup>6</sup>When an `m` script finishes or is closed, all its open streams are also closed. An open stream is also closed when it is no longer referenced and reclaimed by the garbage collector.

<sup>7</sup>`io.stdin` and `io.stdin` represent the same stream; it exists under two different names for historical reasons.

fore twice the number of characters.

- `const utf16be = 3`

Like `utf16le`, but characters are encoded using UTF-16 BE (big endian, high byte first).

## `io.append`

- `function append(path, ces=io.raw) → Native Object`

*Permissions:* `Read+Write(path)`

Opens a file to append to it, and returns its stream object. If the file exists, it is opened for read and write access, and the file pointer is set to its end. If the file doesn't exist, this call is equivalent to `io.create` (p. 114).

If the file already exists, it is truncated to zero length.


Throws `ErrPathNotFound` if the directory does not exist.

```
f=io.append("activity.log");
// file pointer is at the end
print io.size(f), io.seek(f,0,true)
→ 1813 1813
io.close(f)
```

## `io.avail`

- `function avail(stream) → Number`

Returns the number of *bytes* which can be read without blocking. For disk files, this is normally the number of bytes to the end of the file.

For `io.stdin`, this is the number of characters which can be read without changing to input mode, i.e. calling a reading function: console input is normally only accepted during a read on `io.stdin` (when the  state icon is shown). See `ui.keys` (p. 141) for information on removing this restriction.

```
// read all remaining console input
len=io.avail(io.stdin);
s=io.read(io.stdin, len)
```

## io.close

- `function close(stream) → null`

Flushes and closes the file `stream`. Attempts to close `io.stdin` or `io.stdout` are ignored.

See also [io.flush](#) (p. 115).

## io.ces

- `function ces(stream) → Number`
- `function ces(stream, scheme) → Number`

Gets or sets the character encoding scheme of a file. With one argument, returns the current CES of the file `stream`. With two arguments, returns the old CES, and sets the CES of `stream` to `scheme`.

Throws `ErrAccessDenied` when attempting to change the CES of `io.stdin` or `io.stdout`.

## io.create

- `function create(path, ces=io.raw) → Native Object`

*Permissions:* `Write(path)`

Creates a new, empty file in the directory and with the name specified by `path`, and returns its stream object. The initial CES is set to `ces`. The file is opened for read and write access.

If the file already exists, it is truncated to zero length.

Throws `ErrPathNotFound` if the directory does not exist.

```
f=io.create("sample.xml", io.utf8);
print f
→ 2
io.close(f)
```

## `io.flush`

- `function flush(stream) → Boolean`
- `function flush(stream, auto) → Boolean`

With one argument flushes the file `stream`, i.e. writes any pending data to the underlying file or communication stream, and returns the auto flush state.

With two arguments, enables (`auto=true`) or disables (`auto=false`) auto flushing, and returns the previous setting.

If auto flushing is enabled, the file will be flushed after each `io.write...` and `io.print...` call. For optimum performance when writing a lot of data, auto flushing should be disabled.

If a file has auto flushing enabled, calling `io.flush` to flush the file is never required.

By default, auto flushing is enabled.

```
// disable auto flushing before writing a lot of data
old=io.flush(f, false);
for line in lines do
  io.writeln(f, line)
end;
// restore the previous auto flush state
io.flush(f, old)
```

## `io.open`

- `function open(path, rw=false, ces=io.raw) → Native Object`

*Permissions:* `Read(path)` / `Read+Write(path)`

Opens an existing file in the directory and with the name specified by `path`, and returns its stream object. The initial CES is set to `ces`. If `rw=false`, the file is opened for read access, and attempts to write to it will throw `ErrAccessDenied`. If `rw=true`, the file is opened for read and write access.

Throws `ErrPathNotFound` if the directory does not exist, and `ErrNotFound` if the file does not exist.

```
f=io.open("sample.xml", false, io.utf8);
print f
→ stream@41255c
io.close(f)
```

## io.print

- function print(stream, expression, ...) → null

Writes a list of expressions as strings to file `stream`, using the current character encoding scheme. The expressions are converted to strings according to the rules in section 2.7.10 (p. 27). The strings are written one after the other, without separators or a terminator string.

```
old=13;
io.print(io.stdout, "old=", old, ", new: ");
→ old=13, new:
```

## io.println

- function println(stream, expression, ...) → null

Like `io.print`, but also writes a newline (CR and LF characters) after writing all arguments.

## io.read

- function read(stream, len) → String|null

Reads from `stream` until `len` characters have been read, or the file end has been reached, and returns the characters read as a string.

`len` determines the number of characters read, not the number of bytes: with encoding schemes different from `io.raw`, the number of bytes read may be greater than `len`.

Advances the file pointer by the number of bytes read. Returns `null` if the file pointer is already at the end of `stream`. Reading from `io.stdin` never returns `null`, as the user is prompted for new data if there is no data to read.

```
f=io.open("Hello.mp3");
// read first three bytes of MP3 file
print io.read(f, 3);
→ ID3
io.close(f)
```

See also: [.code](#) (p. 45)

## `io.readln`

- function `readln(stream, len=256) → String|null`

Reads from `stream` until `len` characters have been read, or until the next end of line has been reached<sup>8</sup>, and returns the characters read as a string. The string returned does not contain the end of line mark.

`len` determines the number of characters read, not the number of bytes: with encoding schemes different from `io.raw`, the number of bytes read may be greater than `len`.

Advances the file pointer by the number of bytes read. Returns `null` if the file pointer is already at the end of `stream`.

```
f=io.open(system.appdir + "autoexec.m");
// read the first three lines
for i=1 to 3 do
  print io.readln(f)
end
→ /*

    Default autoexec script for interactive shells.

(c) 2005 infowing AG, CH-8703 Erlenbach
io.close(f)
```

## `io.readm`

- function `readm(stream) → anytype`

Reads the next `m` data item from `stream`, and returns it. The data must have

---

<sup>8</sup>end of line is marked by CR-LF, LF, or CR.

been written using `io.witem` (p. 120).

Advances the file pointer by the number of bytes read.

The current encoding scheme does not affect how the input data is interpreted.

Throws `ErrEof` if end of file is reached during reading. Throws `ErrCorrupt` if the data in the file is invalid.

See `io.witem` (p. 120) for an example.

## `io.seek`

- `function seek(stream, pos, current=false) → Number`

Sets the file pointer position of file `stream` to `pos`. If `current=false`, `pos` is an absolute position and must not be negative. If `current=true`, `pos` is relative to the current position and may also be negative.

The file pointer position is always in bytes, independent of the current character encoding scheme.

Returns the new absolute file position.

```
io.seek(f, 0); // seek to beginning of file
io.seek(f, io.size(f)); // seek to end of file
io.seek(f, -40, true); // rewind 40 bytes
current=io.seek(f, 0, true); // get current position
```

## `io.size`

- `function size(stream) → Number`

Returns the size of file `stream`, in bytes.

See also: `files.size` (p. 86)

## `io.timeout`

- `function timeout() → Number`
- `function timeout(ms) → Number`

Gets or sets the timeout used in reads and writes. Without an argument, returns the current timeout in milliseconds. With one argument, returns the



old timeout, and sets the new timeout to `ms`. Setting the timeout to zero (the default) or a negative value disables timeouts, i.e. I/O operations can block indefinitely.

The timeout is used in all following reads and writes: whenever an operation does not complete within the given number of milliseconds, it throws `ErrTimeout`.

```
// give the user three seconds to input data
io.timeout(3000);
try
  s=io.readln(io.stdin)
  // process input
catch e by
  // if it wasn't a timeout, rethrow e
  if index(e, "ErrTimeout") # 0 then throw e end;
  print "You waited too long..."
end
```

## `io.wait`

- function `wait(streams) → Native Object`

Waits until at least one stream in the array `streams` has at least one byte to read from (i.e. `io.avail` (p. 113) returns a value greater than zero), and returns this stream.

`io.wait` is most useful when simultaneously processing several input streams obtained with modules from the extended library (TCP/IP, Bluetooth, IPC), as it avoids the need for a “busy waiting loop”.

```
ipconn=...
btconn=...
case io.wait([io.stdin, ipconn, btconn])
in io.stdin:
  // read from the console
in ipconn:
  // read from ipconn
in btconn:
  // read from btconn
end
```

## io.write

- function write(stream, string) → null

Writes the string `string` to file `stream`, using the current character encoding scheme.

```
f=io.create("sample.txt", io.utf8);
s="un château français";
io.write(f, s);
print len(s), io.size(f)
→ 19 21
```

## io.writeln

- function writeln(stream, string) → null

Writes the string `string`, followed by a newline (CR and LF characters) to file `stream`, using the current character encoding scheme.

```
f=io.create("sample.txt", io.utf8);
s="un château français";
io.writeln(f, s);
print len(s), io.size(f)
→ 19 23
```

## io.writem

- function writem(stream, data) → null

Writes `data` to file `stream`, so it can be read back in via [io.readm](#) (p. 117). `data` can have any `m` type: number, string, boolean, array, or `null`. Function references and native objects can neither be written nor read.

If `data` is an array, elements of it (or its subarrays) which are referenced multiple times are only written once and correctly resolved when they are read back in. This permits to properly write (“serialize”) recursive data structures (which in `m` are always arrays with elements referencing the array itself).

The current encoding scheme does not affect the raw data written.

Throws `ErrArgument` if `data` is of a type which cannot be written.

```
// write a string
data1="Simply a string";
// and a more complex data structure
data2=["One":1, "Two":2.5, false, null, "V":[8,9,10]];
f=io.create("sample.dat");
io.witem(f, data1);
io.witem(f, data2);
io.close(f);
// read it back in
f=io.open("sample.dat");
print io.readm(f)
→ Simply a string
a=io.readm(f);
print a, keys(a)
→ [1,2.5,false,null,Array<3>] [One,Two,null,null,V]
io.readm(f)
→ ErrEof thrown
```

## 3.10 Module `math`: Mathematical Functions

This module provides standard mathematical functions.

### `math.abs`

- function `abs(x) → Number`

Returns the absolute value of `x`.

### `math.acos`

- function `acos(x) → Number`

Returns the arcus cosine (in radians) of `x`.

Throws `ErrArgument` if `abs(x) > 1`.

## **math.asin**

- `function asin(x) → Number`

Returns the arcus sine (in radians) of `x`.

Throws `ErrArgument` if `abs(x) > 1`.

## **math.atan**

- `function atan(x) → Number`

Returns the arcus tangent (in radians) of `x`.

## **math.ceil**

- `function ceil(x) → Number`

Returns the smallest integer greater than or equal to `x`.

```
print math.ceil(3)
→ 3
print math.ceil(3.4)
→ 4
print math.ceil(-3.4)
→ -3
```

## **math.cos**

- `function cos(x) → Number`

Returns the cosine of `x` (in radians).

## **math.exp**

- `function exp(x) → Number`

Returns  $e^x$

## `math.floor`

- function `floor(x) → Number`

Returns the largest integer less than or equal to `x`.

```
print math.floor(3)
→ 3
print math.floor(3.4)
→ 3
print math.floor(-3.4)
→ -4
```

## `math.log`

- function `log(x) → Number`

Returns the natural logarithm of `x`.

## `math.pow`

- function `pow(x, y) → Number`

Returns  $x^y$ .

Throws `ErrArgument` if `x < 0` and `y` is not an integer.

Throws `ErrOverflow` if `x = 0` and `y < 0`.

```
print math.pow(2, 0.5)
→ 1.4142135624
print math.pow(-5, 3);
→ -125
```

## `math.random`

- function `random() → Number`
- function `random(seed) → Number`

Returns a random number uniformly distributed in the interval 0 (inclusive) to 1 (exclusive). With an argument, initializes the sequence of random numbers with `seed`. `seed` can be any number.

The default initialization is based on the current time.

```
math.random(0);  
for i=1 to 3 do  
    print math.random()  
end  
→ 0.0038488093  
   0.6952766137  
   0.2338878537
```

### **math.round**

- function round(x, decimals=0) → Number

Rounds x to decimals decimal digits.

```
print math.round(4.5)  
→ 5  
print math.round(math.pi, 4)  
→ 3.1416
```

### **math.sin**

- function sin(x) → Number

Returns the cosine of x (in radians).

### **math.sqrt**

- function sqrt(x) → Number

Returns the square root of x.

Throws ErrArgument if x < 0.

### **math.tan**

- function tan(x) → Number

Returns the tangent of x (in radians).

## `math.trunc`

- function `trunc(x) → Number`

Returns the integral part of `x`.

```
print math.trunc(3)
→ 3
print math.trunc(3.4)
→ 3
print math.trunc(-3.4)
→ -3
```

## `math` Constants

- const `e` = 2.718281828459045 Euler constant.
- const `pi` = 3.141592653589793  $\pi$ .

## 3.11 Module `sms`: Short Messages

This module supports sending and receiving of short messages.

Messages are identified by numbers. These numbers are used to retrieve and update message contents, and to delete messages.

When a function of the module is called for the first time, it starts listening for incoming messages and enqueues their numbers. Calling `sms.receive` will return these numbers. Messages received earlier can be retrieved from the inbox.

Messages longer than the maximum length (160 characters in the default alphabet) can also be sent and received. They are transmitted as “concatenated SMS”, but the module handles this automatically.

The typical sequence to consume messages starting with a certain token (`//tok` in our example) is:

```

nr=sms.receive(); // wait for a new message
msg=sms.get(nr); // get the message
words=split(msg["text"]); // split the text into words
if len(words)>0 and words[0] = "//tok" then
    // first word is //tok, delete it from inbox
    sms.delete(nr);
    // process message
end

```

## sms.delete

- function delete(msgnum) → null

*Permissions:* WriteApp+FreeComm

Delete the message with number msgnum from the inbox.

Throws ErrNotFound if the message with this number does not exist.

```

// delete all SMS inbox messages older than a week
lastweek=time.get()-7*24*3600;
for id in sms.inbox() do
    if sms.get(id)["time"]<lastweek then
        sms.delete(id)
    end
end
end

```

## sms.get

- function get(msgnum) → Array

*Permissions:* ReadApp+FreeComm

Get the contents of the message with number msgnum. The message contents are returned as an array with the following keys:

Key	Contents
sender	The phone number of the sender of the message.
text	The text of the message.
time	The time stamp of the message, as seconds since the start of year 0. See also module <a href="#">time</a> (p. 131).
unread	true if the message is still unread, false if it has been seen.



Throws `ErrNotFound` if the message with number `msgnum` does not exist.

```
// print all messages in the SMS inbox
for id in sms.inbox() do
  print sms.get(id)
end
→ [248,Delivery confirmation,63277873561,false]
...
```

### `sms.inbox`

- function `inbox()` → Array

*Permissions:* `ReadApp+FreeComm`

Gets the ids of all SMS messages in the inbox.

```
print sms.inbox()
→ [1049241,1049289,1049292]
```

### `sms.receive`

- function `receive(timeout=-1)` → Number|null

*Permissions:* `ReadApp+FreeComm`

Receives a new message and returns its id. If there is no message, waits until one arrives. If `timeout` ≥ 0 and `timeout` milliseconds have passed without receiving anything, returns `null`.

```
// quickly check whether there is a new message
id=sms.receive(0);
if id#null then
  msg=sms.get(id);
  // process msg
end
```

## sms.send

- function send(recipient, message, bits=7) → null
- function send(recipients, message, bits=7) → null

*Permissions:* CostComm

Sends a short message to one or several recipients. A single recipient is specified as a single phone number string, multiple recipients are specified as an array of phone number strings.

bits indicates the number of bits used to encode a character, thus limiting the length of a simple message. Longer messages will be concatenated from several simple messages, thus increasing transmission cost. The allowed values are:

bits	Meaning	Max. length
7	Default text alphabet	160
8	Data alphabet	140
16	Unicode alphabet	70

This function does not return before the message has been sent (or an error occurs).

```
// send a silly message to two people
sms.send(["+41797654321", "+393401234567"],
        "Good morning!")
```

## sms.set

- function set(msgnum, message) → null

*Permissions:* WriteApp+FreeComm

Updates the short message with number msgnum with the fields from message. The keys listed in [sms.get](#) (p. 126) must be used. The sender and text of the message will only be changed in the SMS inbox summary; they cannot be changed in the actual message.

```
// mark all messages in the inbox as unread
for id in sms.inbox() do
  sms.set(id, ["unread":true])
end
```

## 3.12 Module `system`: System Related Functions

This module provides mainly information about the `m` runtime system and the device `m` is running on. Its functions are not guaranteed to be portable, as they are tied to the Symbian OS platform.

### `system.gc`

- `function gc() → Number`

Explicitly request garbage collection, reclaiming unused memory of this process.

### `system.hal`

- `function hal(index) → Number`
- `function hal(index, value) → Number`

Obtain device specific information. With one argument, returns the value of attribute number `index`. With two arguments, sets the value of attribute number `index`, and returns the old value.

Throws `ErrNotSupported` if the attribute cannot be read (or modified).

Please refer to Symbian OS documentation for a complete list of attributes. The following table just lists a few:

Index	Meaning
5	Machine UID
11	CPU frequency in kHz
31	Display width in pixels
32	Display height in pixels
35	Display colors
68	System language: 1=english, 2=french, 3=german, ...
72	System drive: 0=A:, 1=B:, 2=C:, ...

## system.mem

- function mem() → Number
- function mem(expression) → Number

The first form returns the size of memory for data used by `m`, and all its processes. This includes the 60 to 100 kBytes of application memory.

The second form returns the size of memory allocated to `expression`, or what would be reclaimed if `expression` were no longer used.

```
print system.mem()
→ 91984
system.gc(); // collect all garbage
print system.mem(array.create(40, 40, 0))
→ 13964
print system.gc() // reclaim array
→ 13956
```

## system.verbosegc

- function verbosegc() → Number
- function verbosegc(level) → Number

Gets and sets the verbosity level of garbage collection:

0	Garbage collection works silently. This is the default.
1	Whenever garbage collection occurs, a short message with the size of the space reclaimed is printed on the console.
2	Whenever garbage collection occurs, a long message with the size and number of cells of the space in use and the space reclaimed is printed, together with the total data memory in use by <code>m</code> .

```

system.verbosegc(2);
for i=1 to 5 do
    a=array.create(100, 100, 0)
end;
→ GC: used=81K/104, freed=0K/0, total=133K
   GC: used=162K/205, freed=0K/0, total=214K
   GC: used=162K/205, freed=81K/202, total=214K
   GC: used=162K/205, freed=81K/202, total=214K
   GC: used=162K/205, freed=81K/202, total=214K

```

## system Constants

- `const appdir = c:\system\apps\mShell\`

The directory where the `m` application files are installed.

- `const dev = Device (version)`

The device type and, in parentheses, the manufacturer software version. If the device name is just a hexadecimal number (e.g. `0x101fb2ae`), please add a bug report citing this number and indicating the device type.

- `const docdir = c:\documents\mShell\`

The directory where the `m` document files (scripts and module sources) are stored.

- `const os = Symbian`

The operating system of the device.

## 3.13 Module `time`: Time and Date Functions

This module provides access to the real time clock. A given point in time in `m` is always measured as the number of seconds since the beginning of year 0 (assuming the Gregorian calendar).

### `time.dayofweek`

- `function dayofweek(secs=time.get()) → Number`

Gets the day of the week of the point in time defined by `secs`, according to

the following table:

0	Monday
1	Tuesday
2	Wednesday
3	Thursday
4	Friday
5	Saturday
6	Sunday

```
print time.dayofweek()  
→ 0  
print time.dayofweek(time.num('2005-05-13'))  
→ 4
```

## time.get

- function `get()` → Number

Gets the local time in seconds since 0000-01-01 00:00:00. The numeric resolution is down to microseconds, but the actual resolution may be coarser.

```
print time.get()  
→ 63279080895  
print str(time.get(), 1, 4)  
→ 63279080895.9844
```

See also: [.date](#) (p. 45)

## time.set

- function `set(secs)` → null

Sets the local time in seconds since 0000-01-01 00:00:00 to `secs`.

```
time.set(time.get() + 60*60) // advance by 1 hour
```

## `time.num`

- `function num(text, format="YMDhms") → Number`

Converts the string `text` into seconds since 0000-01-01 00:00:00, according to the format `format`.

The format string defines the order of the date and time parts in `text`. Each part finishes if either a character which is not a digit is encountered, or if the part's maximum length is reached. The parts are denoted by the following characters:

Character	Max. length	Meaning
Y	4	Year.
M	2	Month.
D	2	Day.
h	2	Hour (24 hour representation).
m	2	Minute.
s	2	Second.
t	3	Fraction of a second.

One and two digit years are assumed to be in the 21st century, i.e. 2000 is added to them.

Throws `ErrArgument` if `format` contains a character other than those above.

```
print time.get(), time.num(date())
→ 63279080895 63279080895
t=time.num("05-03-27")-40*24*3600;
print time.str(t)
→ 2005-02-15 00:00:00
t=time.num('19:14:18.5', 'hmst')+124.7
print time.str(t,'hh:mm:ss:ttt')
→ 19:16:23.200
```

See also: `time.str` (p. 133)

## `time.str`

- `function str(secs, format="YYYY-MM-DD hh:mm:ss") → String`

Converts the seconds since 0000-01-01 00:00:00 `secs` into a string, according to the format `format`.

Each character in the format string will be converted into a character in the resulting string, according to the following table:

Y	Next digit of year
M	Next digit of month
D	Next digit of day
h	Next digit of hour
m	Next digit of minute
s	Next digit of second
t	Next digit of fractions of second

The format is converted from right to left, except for `t`.

```
print date(), time.str(time.get())
→ 2005-03-14 18:28:15 2005-03-14 18:28:15
print time.str(time.get(), "hh:mm:ss.ttt")
→ 18:28:15.424
print time.str(time.get(), "DD.MM.YY")
→ 14.03.05
```

See also: [time.num](#) (p. 133), [.date](#) (p. 45)

## time.utc

- `function utc() → Number`

Gets the real time in the UTC (Universal Time Coordinate) time zone. This equals Greenwich local time, excluding any shift by daylight saving time.

The difference between local time and UTC time is the local time zone:

```
print time.get() - time.utc()
→ 3600
```

## time.weekofyear

- `function weekofyear(secs=time.get()) → Number`

Gets the week of the year of the point in time defined by `secs`. The first week in the year is the first week having four or more days in the year defined by `secs`.



```
print time.weekofyear()
→ 11
print time.weekofyear(time.num('2005-01-01'))
→ 53
```

## 3.14 Module `ui`: User Interface Functions

This module provides functions to display standard dialogs and menus and to modify the `m` user interface.

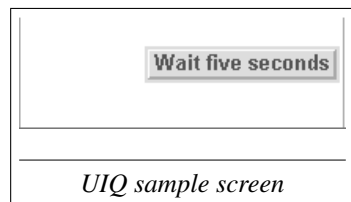
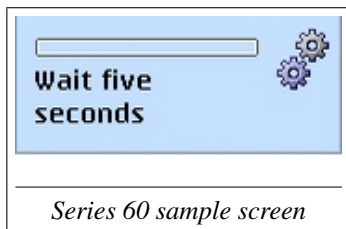
### `ui.busy`

- `function busy(activity) → null`
- `function busy() → null`

With one argument, shows a popup window with the text `activity`, indicating that something is going on. Without an argument, discards the popup window.

Both calls return immediately.

```
ui.busy("Wait five seconds"); // show a popup window
sleep(5000);
ui.busy() // discard the window
```



### `ui.cmd`

- `function cmd(timeout=-1) → Number|String|Array|null`

This function waits for a user command or action:

- A key press, release, or complete keystroke: the function returns the positive scan code for a key press, the negative scan code for a release, or the key code for a keystroke.

For characters, both scan codes and key codes typically correspond to their UNICODE<sup>®</sup> number, and can thus be converted with `.char` (p. 44). Codes for navigation and system keys are device specific. Some important keys are defined as constants (see 3.14 (p. 148)).

`ui.keys` (p. 141) must have been called before to declare interest in such keyboard input.

- A script specific menu command being selected by the user: the function returns the corresponding string from the menu.

`ui.menu` (p. 143) must have been called before to set up the menu.

- The user touches the screen with the pointing device or moves it: the function returns an array with the following elements:

Key	Meaning
x	x-coordinate of pointer
y	y-coordinate of pointer
buttons	mask of pressed buttons: bit 0 for button 1, bit 1 for button 2, bit 2 for button 3.

`ui.ptr` (p. 146) must have been called before to declare interest in such pointer input.

If a monitored user action (keystroke, menu selection, pointing) occurred before `ui.cmd` is called, it immediately returns the corresponding result.

If `timeout>=0` and `timeout` milliseconds have passed without response from the user, `null` is returned.

Keyboard, menu and pointer can all be monitored together in a single `ui.cmd` call.

See `ui.keys` (p. 141) for an example using the keyboard, `ui.menu` (p. 143) for an example using menus, `ui.ptr` (p. 146) for an example using the pointer.

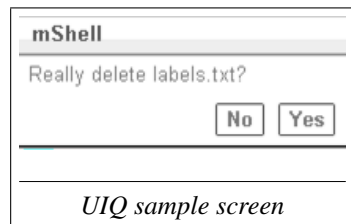
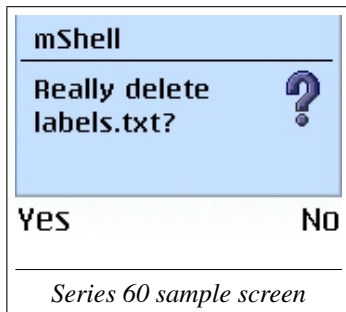
## `ui.confirm`

- `function confirm(question, title="mShell") → Boolean`

Shows a simple dialog displaying `question` in a dialog with title `title`. The dialog asks the user for confirmation, presenting two buttons or soft keys with the options “yes” and “no”.

Returns `true` if the user answers “yes”, and `false` if the user answers “no”.

```
name="labels.txt";
if ui.confirm("Really delete " + name + "?") then
    files.delete(name)
end
```

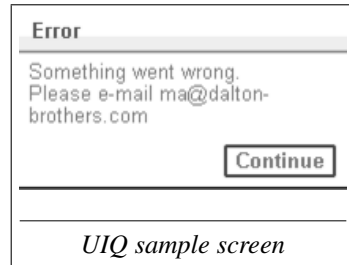


## `ui.error`

- `function error(message) → null`

Displays a dialog with the `error` message, waiting until the user presses the “continue” button or a key.

```
adr="ma@dalton-brothers.com";
ui.error("Something went wrong.\nPlease e-mail " + adr)
```



## ui.fonts

- `function fonts() → Array`

Gets an array with the available fonts. Each font is described by a four element array:

Index	Content	Type
0	Font name	String
1	Minimum font size in pixels	Number
2	Maximum font size in pixels	Number
3	Font is scalable	Boolean

```
print ui.fonts()[0]
→ SwissA,10,19,false
```

## ui.form

- `function form(items, title="mShell") → Array|null`

Displays a dialog to edit the data in `items`, with the given `title`. The keys of `items` will be used as labels (prompts) in the form. Array elements without a key are shown as read-only texts.

The following data types can be edited:

Data Type	Field Type
String without <code>\n</code>	Single line text editor
String with <code>\n</code>	Multi-line text editor <sup>9</sup>
Number	Number editor (floating point)
Boolean	Check box or popup yes/no choice
Array	Combo box or popup multiple choice

The initial values shown in the form are the values given in `items`, except for an array value, where initially the first array element is selected.

If the user presses **Ok**, this function returns an array with the values entered or chosen by the user. If the user presses **Cancel**, `null` is returned.

Setting the `title` is not supported on Nokia devices and silently ignored.

```
old=[ "Name": "",
      "Details:", // just a label
      "Age": 32,
      "Member": false,
      "Beverage": ["Water", "Beer", "Wine", "Whiskey"],
      "Comment": "\n"]; // a multiline field
new=ui.form(old, "Member Card");
print new
→ [Lucky Luke, 35, false, Beer, He's a poor,
    lonesome cowboy]
print keys(new)
→ [Name, Age, Member, Beverage, Comment]
```

Name	<b>Lucky Luke</b>
<b>Details:</b>	
Age	35
Member:	<input type="radio"/> yes <input checked="" type="radio"/> no
Bevera...	Beer
<div>OK</div> <div>Cancel</div>	

*Series 60 sample screen*

Member Card ▾

Name

Details:
 

Age

Member ☐

Beverage  ▾

Comment

Abbr.

OK

*UIQ sample screen*

## ui.idletime

- `function idletime(reset=false) → Number`

Returns the number of milliseconds since the last user activity (keypress or pointer action) on the device. If `reset=true`, resets the inactivity timer to zero.

```
// after about a minute of inactivity, beep
sharp=false;
while true do
  if ui.idletime() < 60000 then
    sharp=true
  elseif sharp then
    audio.beep(); sharp=false
  end;
  sleep(2000)
end
```

## `ui.keys`

- `function keys(pressAndRelease,allowFocus=false) → null`
- `function keys() → null`

Declares interest in keyboard events, for processing by `ui.cmd` (p. 135). Whenever the user performs a keyboard action, the scan code or key code will be returned by the currently waiting or a next call to `ui.cmd`.

If `pressAndRelease=false`, `ui.cmd` will return key codes for complete keystrokes.

If `pressAndRelease=true`, `ui.cmd` will return positive scan codes for key presses and negative scan codes for key releases (each keystroke typically produces two events).

If `allowFocus=true`, the console will obtain the keyboard focus, letting it interpret keystrokes:

- On UIQ devices, the virtual keyboard will be active, and writing a character with the pen will also produce a keystroke.
- On Series 60 devices, the keys will be interpreted as if writing a text.

Keyboard events will be ignored by `ui.cmd` after calling `ui.keys` without arguments.

Each call to `ui.keys` flushes the internal keyboard buffer.

The following example outputs keystrokes until the “go” key is pressed.

```
ui.keys(false); // return keystrokes
do
  c=ui.cmd();
  print "pressed",c,"=",char(c)
until c=ui.gokey
→ pressed 55 = 7
   pressed 42 = *
   pressed 63557 =
```

## ui.large

- `function large() → Boolean`
- `function large(enabled) → Boolean`

Compatibility of function <code>ui.large</code>	
Sony Ericsson phones	UI size change is not possible; function always returns <code>false</code> .

Without arguments, returns the current `m` application view size: `false` if the view size is small (title pane shown), `true` if the view size is large (title pane hidden).

With one argument, return the current view size, and sets the new view size: with `enabled=true`, changes the view size to large, with `enabled=false`, changes the view size to small. This has the same effect as toggling the view size from the menu: it changes the view size for the entire `m` application, in all processes.

## ui.list

- `function list(items, multiple=false, init=[], title="mShell") → Array|null`

Displays a list dialog to choose from the data in `items`:

- If `multiple=false`, only one item can be selected. This is usually simply the highlighted (current) item.
- If `multiple=true`, multiple items can be selected. These are usually the marked items.

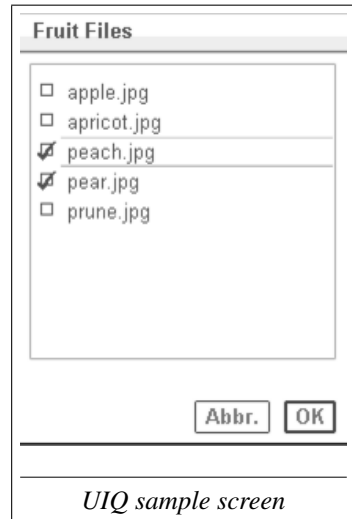
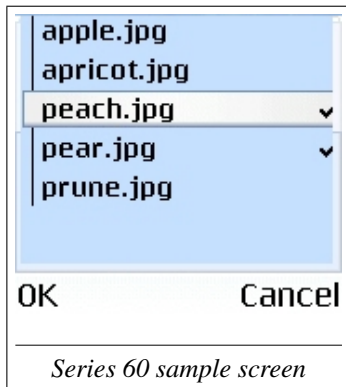
Initially, the items indexed in `init` will be selected (or marked).

If the user presses “ok”, this function returns the indices of the items selected by the user, i.e. an array of numbers indexing into `items`. If the user presses “cancel”, `null` is returned.

`title` is not supported on Nokia devices and silently ignored.



```
f=["apple.jpg", "apricot.jpg", "peach.jpg",
  "pear.jpg", "prune.jpg"];
print ui.list(f, true, [1,3], "Fruit Files")
→ [2,3]
```



## `ui.menu`

- `function menu(title, commands, keepold=true, interrupt=false) → null`
- `function menu() → null`

Replace the standard “Process” menu by a new menu, with `title` and the menu items defined by array `commands`, for processing by `ui.cmd` (p. 135).

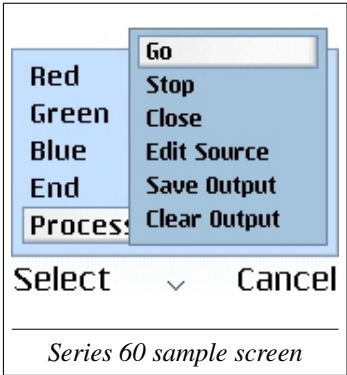
If `keepold=true`, the standard process menu will be added at the end, as a submenu. If `keepold=false`, the standard functions are not available, preventing the user from easily stopping or closing the running process.

If `interrupt=true`, a menu selection by the user will interrupt a waiting function call (except `ui.cmd`) with `ExcInterrupted`. If `interrupt=false`, function calls will not be interrupted, and the menu selection will go unnoticed until `ui.cmd` is called.

Without arguments, restores the standard menu.

Whenever the user selects a menu item, the item will be returned by the currently waiting or the next call to `ui.cmd` (p. 135).

```
ui.menu("Colors", ["Red", "Green", "Blue", "End"]);
while true do
  c=ui.cmd();
  if c="End" then break end;
  print c,"chosen"
end
```



**ui.mfont**

- function `mfont()` → Array
- function `mfont(font)` → Array

Gets or sets the font used in all m consoles. Without parameter, returns the currently used font as an array with the following elements:

Index	Meaning	Type
0	Font name	String
1	Font size in pixels	Number
2	Bold font	Boolean
3	Italic font	Boolean

If the parameter `font` is a string, set the font to the one with the given name, without changing the other attributes.

If the parameter `font` is an array, the array must have the elements listed above, and the font is set accordingly.

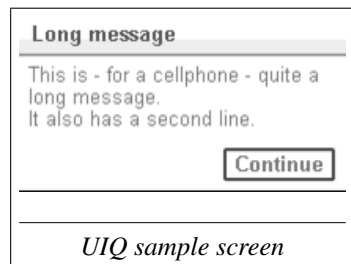
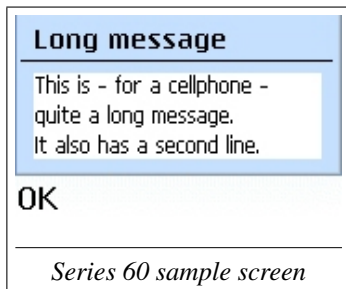
```
old=ui.mfont();
print old
→ [Monospace,11,false,false]
// use a proportional sans serif font
ui.mfont("SwissA");
// make it large and bold
ui.mfont(["SwissA", 16, true, false])
```

## `ui.msg`

- function `msg(message, title="mShell")` → null

Displays a dialog with `message`, waiting until the user presses the “continue” button or a key. `message` can have multiple lines, separated by `\n` characters.

```
ui.msg
("This is - for a cellphone - quite a long message."
 + "\nIt also has a second line.",
 "Long message");
```



## ui.pfonts

- `function pfonts() → null`

Prints a table of the available fonts, with the following columns:

- Font name.
- Minimal and maximal size in pixels, separated by `-`.
- Number of scaling steps from minimal to maximal size, prefixed by `x`.
- Font attributes: `p`: proportional, `s`: serif, `y`: symbol, `S`: scalable.

```
ui.pfonts()
→ SwissA      10-19x4 p---
  Courier      8- 8x1 -s--
  Symbol      11-16x2 p-y-
  Calc        13-35x3 --y-
  Eikon       15-15x1 --y-
  Calcinv     14-14x1 --y-
  Digital     35-35x1 --y-
```

## ui.ptr

- `function ptr(absoluteCoord) → null`
- `function ptr() → null`

Declares interest in pointer events, for processing by `ui.cmd` (p. 135). Whenever the user performs a pointing device action, the pointer coordinate and button will be returned by the currently waiting or a next call to `ui.cmd`.

To generate these events, there must be a pointing device: on UIQ devices, the pen corresponds to button one. However, unlike a mouse, the pen only generates events while button is pressed, i.e. the pen touches the screen<sup>10</sup>.

If `absoluteCoord=true`, `ui.cmd` will return absolute coordinates (the origin is the upper left corner of the screen).

If `absoluteCoord=false`, `ui.cmd` will return relative coordinates (the origin is the upper left corner of the console, or graph view).

<sup>10</sup>mVNC has limited support for the Series 60 pointer via the mouse.

Pointer events will be ignored by `ui.cmd` after calling `ui.ptr` without arguments.

The following example outputs the position of the pointing device, until the pen goes up (button one is no longer pressed) in the upper left corner of the console.

```
ui.ptr(false); // return relative coordinates
do
  c=ui.cmd();
  print "at",c["x"],c["y"]
until c["x"]<=10 and c["y"]<=10 and c["buttons"]=0
→ at 123 116
   at 123 146
   at 91 142
   ...
   at 11 7
   at 8 7
   at 7 7
```

## `ui.query`

- `function query(prompt, title="mShell", value="") → String|Number|null`

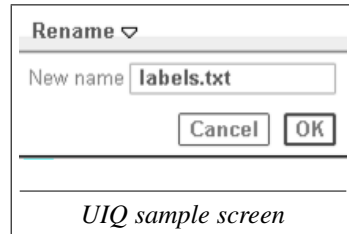
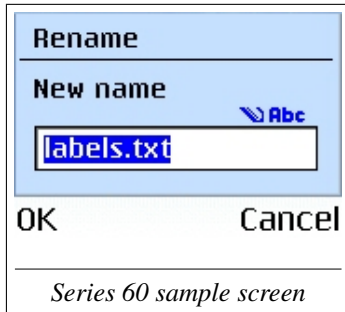
Displays a dialog querying for a single text input. The input field is initialized with `value`, and labelled with `prompt`.

If `value` is a number, the input field is numeric and does not allow non-numeric characters. The only valid characters are `0123456789-+,.Ee`. The return value will also be numeric in this case. The function throws `ExcInvalidNumber` if the format of the number entered is not valid.

If the user presses “ok”, this function returns the value entered by the user. If the user presses “cancel”, `null` is returned.

The same effect can be achieved with `ui.form` (p. 138), but `ui.query` is simpler to use.

```
old="labels.txt";
new=ui.query("New name", "Rename", old);
if new#null and new#old then
    files.rename(old, new)
end
```



## ui Constants

These constants define the key codes (for keystrokes) of the navigation keypad typically found on Nokia phones, and the Jog Dial on Sony Ericsson phones.

- const **downkey** = *down key code* The “down” navigation key.
- const **gokey** = *go key code* The “go” or “confirm” navigation key.
- const **leftkey** = *left key code* The “left” navigation key.
- const **rightkey** = *right key code* The “right” navigation key.
- const **upkey** = *up key code* The “up” navigation key.

## 4. Interactive Shells

`m` cannot only execute complete scripts, it can also be used interactively, as a shell. When working in shell mode, there are a few differences to normal `m` scripts:

- `m` statements are executed interactively: `m` code can be entered and is executed immediately. Global variables and functions are preserved between executions.
- The syntax allows some simplifications (see section 4.1 (p. 149)).
- Each time a shell is created, it loads and executes `autoexec.m` before prompting the user. The script is first searched among the ordinary scripts in `system.docdir` (p. 131). If it is not found, the default script in `system.appdir` (p. 131) is executed.

### 4.1 Simplified Syntax for Interactive Use

Since input capabilities of cellphones are poor, interactive shells support a simplified syntax for function calls, and automatic output of computed expressions:

- A single `Expression` will be executed as `'print' Expression`, unless it is `null`:

```
m>0.85*23.10
→ 19.635
m>use math as m
m>m.sin(m.pi/4)
→ 0.7071067812
```

- A `SimpleFunctionCall` calls a function with only string or number literal parameters, and options defined for the function.

- Unquoted words (sequences not containing white space) on the command line which are not keywords (see appendix A.2 (p. 161)) and are not starting with a digit or separator are interpreted as string parameters.
- Numbers are interpreted as numeric parameters.
- Options for optional parameters (see section 2.8 (p. 29)) can be specified anywhere with a preceding slash. If an equal sign follows, the following word or number is assigned to the corresponding parameter. If no equal sign follows, `true` is assigned to the corresponding parameter.
- Commas to separate the parameters are not permitted.

Again, the function result is printed if it is not null:

```
m>date // maps to date()
→ 2005-02-07 11:03:07
m>dir c:\*.m/r // maps to dir('c:\*.m', true)
→ C:\system\apps\mShell\autoexec.m
   C:\documents\mShell\Jukebox.m
```

Simple function calls can only be used to call functions with parameters which are string or number literals.

```
SimpleFunctionCall :=
  [ModulePrefix] Identifier {SimpleParam | SimpleOption} .
SimpleParam :=
  SimpleChar {SimpleChar} | StringLiteral | NumberLiteral .
SimpleOption := '/' (IdentifierChar | Digit) ['=' SimpleParam] .
SimpleChar :=
  (printable ISO-8859-1 char except white space and '/') .
```

## 4.2 Shell Builtin Functions

`autoexec.m` defines a number of function for interactive use. Most are just wrappers around existing functions, to avoid typing longer names. With these functions, files on the phone can be easily manipulated:



```
// list all JPG files on the current drive
dir \*.jpg/r/l
→ -- 05-08-09 2966 \documents\mShell\GraphTest.jpg
   -- 11:37:02 17909 \Nokia\Images\FE_img\FEscr(0).jpg
...
// copy the JPG files in \documents\mShell to drive e:
cp \documents\mShell\*.jpg e:
→ 1
// search for the mShell properties file
dir \*.prp/r
→ \System\Apps\mShell\mShell.prp
// show its contents
type \system\apps\mShell\mShell.prp
→ mfont=LatinPlain12
   outsize=20000
   keep=busy
```

If a customized `autoexec.m` in `system.docdir` is created without incorporating the original script, these function are no longer available.



### **.cp**

- function `cp(src, dst, recursive=false) → Number`  
`/r:recursive`

Copies a file, files matching a pattern, or an entire directory tree. Wrapper for `files.copy` (p. 80).

### **.del**

- function `del(pattern, recursive=false) → Number`  
`/r:recursive`

Deletes a file, files matching a pattern, also in complete directory tree. Wrapper for `files.delete` (p. 80).

### **.dir**

- `function dir(pattern="*", recursive=false, long=false, hidden=false, modified=0) → null`  
    `/h:hidden`  
    `/l:long`  
    `/m:modified`  
    `/r:recursive`

List files matching `pattern` on standard output. If `pattern` is a directory, lists all files in it. Options are the following:

- With `/h (hidden=true)`, also lists hidden files and directories.
- With `/l (long=true)`, lists files and directories in a long format, including `readonly` and `hidden` attributes and modification date (format `YY-MM-DD` or `hh:mm:ss`).
- With `/m=secs (modified=secs)`, lists only files which were modified within the last `secs` seconds.

### **.edit**

- `function edit(name) → null`

Loads a file into the builtin editor and shows it. Wrapper for `files.edit` (p. 81).

### **.exit**

- `function exit() → null`

Exit this shell. This is equivalent to closing it. This function is only available if module `proc` is available.

### **.md**

- `function md(path, all=false) → Number`  
    `/a:all`

Creates a directory or directories. Wrapper for `files.mkdir` (p. 82).

### **.mv**

- `function mv(src, dst, recursive=false) → Number`  
    `/r:recursive`

Moves a file, files matching a pattern, or an entire directory tree. Wrapper for `files.move` (p. 82).

### **.rd**

- `function rd(path, recursive=false) → Number`  
    `/r:recursive`

Removes a directory or an entire directory tree. Wrapper for `files.rmdir` (p. 83).

### **.ren**

- `function ren(old, new) → Number`

Renames a single file. Wrapper for `files.rename` (p. 83).

### **.run**

- `function run(script, show=false) → null`  
    `/s:show`

Run another m script. If `show=true`, the script's console is shown. This function is only available if module `proc` is available.

### **.send**

- `function send(name, subject=null) → null`

Interactively sends a file over a channel chosen by the user. Wrapper for `files.send` (p. 85).

## `.type`

- `function type(file, utf16=false, tail=false) → null`  
    /u:utf16  
    /t:tail

Writes the contents of `file` to standard output.

If `utf16=true`, assumes the file to be UTF-16 little endian encoded. Otherwise, raw encoding is assumed.

If `tail=true`, only outputs the last 300 bytes. If `tail=n` where `n` is a number, outputs the last `n` bytes.

## 5. SMS Control

If the *m* Supervisor & Viewer application is installed and licensed, the *m* application can be controlled via SMS commands. Commands must be prefixed by the *smskey* configured in the properties (see section A.3 (p. 161)).

The available SMS commands are:

- *smskey run script args*: starts the *m* application if it is not already running, then starts the script *script* with the arguments *args*. Use function *proc.args* to get the arguments from within the script. If the script is already running, this command is ignored.
- *smskey shutdown*: stops all scripts and exits the *m* application. If *m* is not running, this command is ignored.
- *smskey start*: starts the *m* application. If *m* is already running, this command is ignored.
- *smskey status*: *m* status inquiry, replies with an SMS describing the status of the *m* application and some GSM information. If *m* is running, the reply will look like:

```
m status: running, mem=mem,  
net=mcc,mnc, loc=lac,cid, sig=sig
```

If *m* is not running, the reply will look like:

```
m status: NOT running (category reason),  
net=mcc,mnc, loc=lac,cid, sig=sig
```

The meaning of the fields is the following:

<code>mem</code>	bytes of memory used by <code>m</code>
<code>category</code>	<code>m</code> exit category (if panicked)
<code>reason</code>	<code>m</code> exit reason (if panicked)
<code>mcc</code>	GSM mobile country code
<code>mnc</code>	GSM mobile network code
<code>lac</code>	GSM location area code
<code>cid</code>	GSM cell id
<code>signal</code>	GSM signal strength

- `smskey status phone`: like `status` above, but the response is sent to phone number `phone`. `phone` must not contain white space.
- `smskey stop script`: stops execution of script `script`. If `script` is not running, this command is ignored.

The following examples require the `smsctrl` property to be enabled, and `smskey` to be set to `mshell`:

1. SMS to start the `m` application:

```
mshell start
```

2. SMS to start the `Supervisor` script, passing it 0769988776 as an argument:

```
mshell run Supervisor 0769988776
```

3. SMS to check the status of the `m` application:

```
mshell status
→ m status: NOT running (E32USER-CBase 71),
    net=228,115, loc=1616,17689, sig=3
```

`m` is not running because it crashed with a `E32USER-CBase 71` panic. The phone is somewhere near cell 17689 in area 1616 of the Swisscom GSM network.

# A. Appendix

## A.1 Exception Tags

This section lists the exceptions tags with their english error message.

### Environment Exceptions

Environment exceptions are usually thrown by the underlying operation system, e.g. when trying to access a file which does not exist.

- `ErrAbort`: Operation aborted.
- `ErrAccessDenied`: Access denied.
- `ErrAlreadyExists`: File already exists.
- `ErrArgument`: Invalid function argument.
- `ErrBadHandle`: Object handle is bad.
- `ErrBadName`: Name is bad.
- `ErrCancel`: Operation canceled.
- `ErrCommsFrame::` Serial framing error.
- `ErrCommsLineFail::` Serial line failed.
- `ErrCommsOverrun::` Serial overrun error.
- `ErrCommsParity::` Serial parity error.
- `ErrCorrupt`: File or database corrupted.
- `ErrCouldNotConnect::` Could not connect.
- `ErrCouldNotDisconnect::` Could not disconnect.
- `ErrDied`: Thread or process died.
- `ErrDirFull`: Directory is full.

- `ErrDisconnected::` Link is disconnected.
- `ErrDiskFull`: Disk is full.
- `ErrDivideByZero`: Integer division by zero.
- `ErrEof`: Eof reached.
- `ErrGeneral`: General problem.
- `ErrHardwareNotAvailable`: Hardware is not available or not enabled.
- `ErrInUse`: File or device is in use.
- `ErrLocked`: Object locked.
- `ErrNoMemory`: Out of memory. *This exception cannot be caught.*
- `ErrNotFound`: File or item not found.
- `ErrNotReady`: Device is not ready.
- `ErrNotSupported`: Operation not supported.
- `ErrOverflow`: Numeric overflow.
- `ErrPathNotFound`: Path not found.
- `ErrTimedOut::` Operation timed out.
- `ErrTooBig::` Value or array too big.
- `ErrTotalLossOfPrecision`: Total loss of precision.
- `ErrUnderflow`: Numeric underflow.
- `ErrWrite`: Write failed.
- `ExcNotPermitted`: Operation not permitted by user.

## Programming Exceptions

Programming exceptions are thrown by `m`, and usually caused by an error in your code or an unexpected user input.

- `ExcArrayNotNumber`: Operand is an array, not a number.
- `ExcBooleanNotNumber`: Operand is a boolean, not a number.
- `ExcForwardFunction`: Function is only forward defined.



- **ExcFunctionNotNumber:** Operand is a function, not a number.
- **ExcIndexOutOfRange:** Array index is out of range.
- **ExcInterrupted:** Interrupted function call.
- **ExcInvalidIndexType:** Array index is neither number nor string.
- **ExcInvalidNumber:** Wrong number format.
- **ExcInvalidUTF8:** Invalid UTF-8 character read.
- **ExcNativeNotNumber:** Operand is native object, not a number.
- **ExcNoSuchKey:** No array element for key.
- **ExcNotArray:** Operand is not an array.
- **ExcNotAvailable:** Function or variable is unavailable.
- **ExcNotBoolean:** Operand is not a boolean.
- **ExcNotComparable:** Can only order two numbers or two strings.
- **ExcNotFunction:** Operand is not a function reference.
- **ExcNotNative:** Operand is not a native object.
- **ExcNotNumber:** Operand is not a number.
- **ExcNotString:** Operand is not a string.
- **ExcNullNotNumber:** Operand is null, not a number.
- **ExcStringNotNumber:** Operand is a string, not a number.
- **ExcStringPosOutOfRange:** String position is out of range.
- **ExcTooManyGlobals:** Too many global variables, split into modules.
- **ExcUnknownModule:** Unknown module referenced by native function.
- **ExcValueOutOfRange:** Value or parameter is outside valid range.
- **ExcWrongNative:** Operand has wrong native object type.
- **ExcWrongParamCount:** Too many or too few function parameters.

## Internal Error Exceptions

Internal error exceptions are thrown by `m` when it detects an internal inconsistency. These exceptions cannot be caught, and are most likely caused by a bug in `m` or in a native module.

- `ErrDisabledFunction`: Internal error: interpreting disabled function.
- `ErrDuplicateModule`: Internal error: duplicate module.
- `ErrDuplicateNative`: Internal error: duplicate native function.
- `ErrEndOfCode`: Internal error: falling through end of code.
- `ErrInvalidDll`: Internal error: DLL did not return module.
- `ErrInvalidFrame`: Internal error: invalid stack frame contents.
- `ErrInvalidInstruction`: Internal error: invalid instruction.
- `ErrInvalidStack`: Internal error: invalid stack.
- `ErrInvalidVariableIndex`: Internal error: invalid variable index.
- `ErrMissingDll`: Internal error: module DLL is missing.
- `ErrNativeFunction`: Internal error: interpreting native function.
- `ErrNoCode`: Internal error: interpreting without code.
- `ErrNoNativeFunction`: Internal error: no native function to add option to.
- `ErrRTVersionMismatch`: Internal error: runtime version mismatch.  
*Get an up to date version of the runtime or native module.*
- `ErrStringExtension`: Internal error: string extension.

## A.2 Reserved words

In the `m` language, keywords, like identifiers, are case sensitive. The following keywords are reserved and cannot be used as identifiers:

<code>and</code>	<code>do</code>	<code>function</code>	<code>shl</code>	<code>until</code>
<code>as</code>	<code>else</code>	<code>if</code>	<code>shr</code>	<code>use</code>
<code>break</code>	<code>elsif</code>	<code>in</code>	<code>then</code>	<code>while</code>
<code>by</code>	<code>end</code>	<code>not</code>	<code>throw</code>	
<code>case</code>	<code>false</code>	<code>null</code>	<code>to</code>	
<code>catch</code>	<code>for</code>	<code>or</code>	<code>true</code>	
<code>const</code>	<code>forward</code>	<code>return</code>	<code>try</code>	

## A.3 Properties (.prp) File

Global behaviour of the `m` application is configured in the `m` properties. Selecting **View**→**Properties** opens a dialog to edit the properties.

The properties are stored in an ASCII text file `\system\apps\mShell\mShell.prp` containing key-value pairs. Each pair is on a single line, the key and the value separated by an equal (=) character.

The following keys are recognized by `m`:

- `autogo=script1,script2,...`  
A comma separated list of scripts to run when starting `m`. In conjunction with `onboot`, these scripts are run when the phone is switched on. The script names must not contain any blanks.
- `bgcolor=black|white|red|green|blue|yellow|cyan|magenta|#rrggbb`  
The background color of console and editor. `#rrggbb` is a HTML-like hexadecimal notation, e.g. `#ff00ff` for magenta.
- `encodings=bom|utf-8|utf-16le|utf-16be|8-bit`  
The encoding to use for `m` source files and files loaded into and saved from the `m` editor. This setting does *not* change the behaviour of the I/O streams of module [io](#) (p. 111).  
If set to `bom`, files read are expected to carry an initial Byte Order Mark

(BOM, character `0xfeff`) determining their encoding; files without BOM are treated as sequences of 8-bit characters. In this mode, files are saved in UTF-8 with initial BOM.

If set to `utf-8`, files are read and saved in UTF-8. No BOM is expected or written.

If set to `utf-16le`, files are read and saved in UTF-16 Little Endian. No BOM is expected or written.

If set to `utf-16be`, files are read and saved in UTF-16 Big Endian. No BOM is expected or written.

If set to `8-bit`<sup>1</sup>, files are read and saved considering only the lower eight bits of all characters. No BOM is expected or written.

- `fgcolor=black|white|red|green|blue|yellow|cyan|magenta|#rrggbb`

The foreground (text) color of console and editor.

- `keep=true|yes|y|1 | false|no|n|0 | busy`

If set to `true`, `yes`, `y` or `1`, the `m` application cannot be exited automatically by the system, e.g. if it is running low on memory, or if `m` is to be removed because it is updated by a new installation.

If set to `busy`, exiting is prevented if there are processes running or waiting for input.

For all other values, `m` behaves like any other “well behaving” application, i.e. it can be exited at any time if the operating system requests it.

- `mfont=typeface,points,bold,italic`

The font to use in the `m` console and editor. `points` (integer), `bold` (boolean) and `italic` (boolean) are optional. See also [ui.mfont](#) (p. 144).

- `onboot=true|yes|y|1 | false|no|n|0 | once | restart`

If set to `true`, `yes`, `y` or `1`, the `m` application will be started automatically when the phone is booted up, i.e. switched on.

If set to `once`, `m` is only started at the next bootup, as the entry is automatically set to `n` afterwards. This is the recommended setting for disaster prevention during script testing.

If set to `restart`, `m` is started automatically when the phone is booted

---

<sup>1</sup>This was the mode used prior to version 1.17.

up, and restarted each time about 20 seconds after it exits (orderly or because of a crash).

This feature requires the *m* Supervisor & Viewer application to be installed.

- `outsize=charcount`  
The maximum number of characters in the console output, before truncating at the beginning. Truncation happens in chunks of about 500 characters. Set to 0 for an unlimited output size. Handling large output output sizes slows *m* down.
- `perms=permissions`  
The permission bits, defining the permissions granted to *m* scripts. See section A.4 (p. 165).
- `smsctrl=true|yes|y|1 | false|no|n|0`  
If set to *true*, *yes*, *y* or *1*, the *m* application can be controlled via SMS commands, even if it is not running. See chapter 5 (p. 155).  
This feature requires the *m* Supervisor & Viewer application to be installed.
- `smskey=keyword`  
Any SMS containing *keyword* as the first characters (ignoring case) is considered a command and sent to the *m* application.
- `smsnr=suffix`  
The last digits of the sender phone number which can control the *m* application via SMS. If empty, anybody knowing `smskey` can control *m*.



All other keys are silently ignored. This can be used to disable entries by just putting e.g. a hash mark in front of them.

A sample properties file might look as follows:

```
autogo=TrackMe,PhoneMonitor
keep=busy
mfont=Monospace,14,false,false
onboot=once
fgcolor=#008000
bgcolor=white
outsize=10000
encoding=utf-8
perms=159
smsctrl=yes
smsnr=4561234
smskey=mshell
```

## A.4 User Permissions

Permission for certain operations can be granted and denied by the user. Any operation with insufficient permissions will throw `ExcNotPermitted`. Selecting **View**→**Permissions** opens a dialog to edit the permissions.

The individual permissions are:

Name	Bit	Meaning
ReadDoc	1	Read access to files in <code>system.docdir</code> and its sub-directories.
WriteDoc	2	Write access to files in <code>system.docdir</code> and its sub-directories.
ReadApp	4	Read access to other application's data.
WriteApp	8	Write access to other application's data.
FreeComm	16	Access to free communication resources (receiving messages, Bluetooth).
ReadAll	32	Read access to all files.
WriteAll	64	Write access to all files. <b>Granting write access to all files also allows modifying the permissions.</b>
CostComm	128	Access to chargeable communication resources (sending messages, TCP/IP).

Thus, if a function requires `Read(file)`, then

- If `file` denotes a file or directory in `system.docdir` or one of its sub-directories, the `ReadDoc` permission must be granted for the function to succeed.
- If `file` denotes a file or directory outside `system.docdir` or one of its subdirectories, the `ReadAll` permission must be granted for the function to succeed.

Likewise, if a function requires `Write(file)`, the `WriteDoc` or `WriteAll` permissions must be granted.





# Index

- ..., 36
- .prp file, 161
- ;, 18
- 8-bit, 162
- abs function (in math), 121
- acos function (in math), 121
- add function (in contacts), 72
- adr, contact field, 71
- alaw constant (in audio), 66
- all constant (in files), 79
- appdir constant (in system), 131
- append function (builtin), 43
- append function (in io), 113
- arch constant (in files), 79
- Array, 6
- array
  - associate, 12
  - indexing, 11
  - key, 12
  - literal, 11
- array module, 55
- arrays, 11
- asin function (in math), 122
- assignment, 19
- atan function (in math), 122
- attr function (in files), 79
- attribute bits, 79
- au constant (in audio), 66
- AU format, 63, 66
- audio file, 63
- audio module, 63
- auto flushing, 115
- autoexec.m, 149–151
- avail function (in io), 113
- background color, 89, 90
- beep function (in audio), 63
- bg function (in graph), 90
- birth, contact field, 71
- black constant (in graph), 88
- blue constant (in graph), 88
- Bluetooth, 85
- BMP, 98
- bom, 161
- Boolean, 6
- boolean
  - literal, 9
- break, 27
- brush color, 89, 90
- brush function (in graph), 90
- Builtin Functions and Constants, 43
- busy function (in audio), 64
- busy function (in ui), 135
- buttons, pointer event field, 136
- case, 25
- cd function (builtin), 43
- ceil function (in math), 122

- cell, contact field, 71
- CES, 112
- ces function (in io), 114
- char function (builtin), 44
- character encoding scheme, 112
- cid function (in gsm), 109
- circle function (in graph), 91
- clear function (in graph), 92
- clone, 30
- close function (in audio), 64
- close function (in io), 114
- cls function (builtin), 44
- cmd function (in ui), 135
- code function (builtin), 45
- collate constant (in array), 63
- collate function (builtin), 45
- comments, 6
- company, contact field, 71
- concat function (in array), 55
- concatenation, 16
- confirm function (in ui), 137
- console input, 113
- console mode, 87, 95
- const, 20
- constant, 20
- contact
  - database, 70
  - fields, 71
- contacts, 70
- contacts module, 70
- copy function (in array), 56
- copy function (in files), 80
- cos function (in math), 122
- CostComm, 165
- country, contact field, 71
- cp function (autoexec.m), 151
- create function (in array), 56
- create function (in io), 114
- current directory, 42
- cut function (in audio), 65
- cyan constant (in graph), 88
- data types, 5
- date function (builtin), 45
- dayofweek function (in time), 131
- del function (autoexec.m), 151
- delete function (builtin), 46
- delete function (in contacts), 73
- delete function (in files), 80
- delete function (in sms), 126
- dev constant (in system), 131
- dialogs, 135
- dir constant (in files), 79
- dir function (autoexec.m), 152
- do, 23
- docdir constant (in system), 131
- double dot, 35, 36
- down constant (in graph), 108
- downkey constant (in ui), 148
- dtmf function (in audio), 65
- e constant (in math), 125
- e-mail, 85
- edit function (autoexec.m), 152
- edit function (in files), 81

- ellipse function (in graph), 92
- email, contact field, 71
- equal function (builtin), 46
- ErrAbort, 157
- ErrAccessDenied, 65, 66, 114, 115, 157
- ErrAlreadyExists, 157
- ErrArgument, 65, 68, 74–76, 120–124, 133, 157
- ErrBadHandle, 157
- ErrBadName, 72, 76, 157
- ErrCancel, 157
- ErrCommsFrame:, 157
- ErrCommsLineFail:, 157
- ErrCommsOverrun:, 157
- ErrCommsParity:, 157
- ErrCorrupt, 118, 157
- ErrCouldNotConnect:, 157
- ErrCouldNotDisconnect:, 157
- ErrDied, 157
- ErrDirFull, 157
- ErrDisabledFunction, 160
- ErrDisconnected:, 158
- ErrDiskFull, 158
- ErrDivideByZero, 158
- ErrDuplicateModule, 160
- ErrDuplicateNative, 160
- ErrEndOfCode, 160
- ErrEof, 118, 158
- ErrGeneral, 158
- ErrHardwareNotAvailable, 158
- ErrInUse, 63–66, 68, 69, 158
- ErrInvalidDll, 160
- ErrInvalidFrame, 160
- ErrInvalidInstruction, 160
- ErrInvalidStack, 160
- ErrInvalidVariableIndex, 160
- ErrLocked, 158
- ErrMissingDll, 160
- ErrNativeFunction, 160
- ErrNoCode, 160
- ErrNoMemory, 158
- ErrNoNativeFunction, 160
- ErrNotAvailable, 37, 38
- ErrNotFound, 73, 75, 77, 115, 126, 127, 158
- ErrNotReady, 65, 68, 158
- ErrNotSupported, 65, 69, 104, 108, 129, 158
- error function (in ui), 137
- ErrOverflow, 123, 158
- ErrPathNotFound, 113–115, 158
- ErrRTVersionMismatch, 160
- ErrStringExtension, 160
- ErrTimedOut, 119
- ErrTimedOut:, 158
- ErrTooBig:, 158
- ErrTotalLossOfPrecision, 158
- ErrUnderflow, 158
- ErrWrite, 158
- ExcArrayNotNumber, 158
- ExcBooleanNotNumber, 158
- ExcDivideByZero, 14
- exceptions, 38
  - catching, 39
  - environment, 157
  - internal, 160

- programming, 158
- tags, 157
- throwing, 39
- ExcForwardFunction, 158
- ExcFunctionNotNumber, 159
- ExcIndexOutOfRange, 11, 39, 56–59, 61, 159
- ExcInterrupted, 143, 159
- ExcInvalidIndexType, 159
- ExcInvalidNumber, 147, 159
- ExcInvalidParam, 72
- ExcInvalidUTF8, 112, 159
- ExcNativeNotNumber, 159
- ExcNoSuchKey, 61, 159
- ExcNotArray, 159
- ExcNotAvailable, 159
- ExcNotBoolean, 22, 23, 159
- ExcNotComparable, 17, 59, 62, 159
- ExcNotFunction, 159
- ExcNotNative, 159
- ExcNotNumber, 159
- ExcNotPermitted, 158, 165
- ExcNotString, 159
- ExcNullNotNumber, 159
- ExcStringNotNumber, 159
- ExcStringPosOutOfRange, 46, 47, 51, 54, 159
- ExcTooManyGlobals, 159
- ExcUnknownModule, 159
- ExcValueOutOfRange, 64, 74, 159
- ExcWrongNative, 159
- ExcWrongParamCount, 159
- exists function (in files), 81
- exit function (autoexec.m), 152
- exp function (in math), 122
- expressions, 13
- extadr, contact field, 71
- extname, contact field, 71
- fax, contact field, 71
- file
  - attribute, 79, 84
  - name, 41
- files module, 78
- fill function (in array), 57
- find function (in contacts), 73
- findnr function (in contacts), 74
- floor function (in math), 123
- flush function (in io), 115
- fname, contact field, 71
- fold constant (in array), 63
- font, 93
- font function (in graph), 93
- fonts function (in ui), 138
- for, 23
- form function (in ui), 138
- forward, 32
- FreeComm, 165
- full function (in graph), 94
- full screen mode, 87, 94, 96
- function
  - forward, 32
  - literal, 9
  - parameter, 11, 30
  - recursive, 11, 30
  - reference, 6, 29, 33, 48

- result, 30
- function reference, 33
- functions, 29
- garbage collection, 129, 130
- gc function (in system), 129
- get function (in contacts), 75
- get function (in graph), 97
- get function (in sms), 126
- get function (in time), 132
- GIF, 98
- GIF format, 104
- Global variables, 11
- gokey constant (in ui), 148
- graph module, 87
- graphics, 87
  - colors, 88
  - coordinates, 87
- green constant (in graph), 88
- gsm module, 108
- hal function (in system), 129
- hexadecimal, 8
- hexnum function (builtin), 47
- hexstr function (builtin), 47
- hidden constant (in files), 79
- hide function (in graph), 98
- icon function (in graph), 98
- idletime function (in ui), 140
- if, 21
- ima constant (in audio), 66
- imei constant (in gsm), 111
- imsi constant (in gsm), 111
- inactivity timer, 140
- inbox function (in sms), 127
- increment, 20
- index function (builtin), 47
- index function (in array), 57
- insert function (in array), 58
- io module, 111
- isarray function (builtin), 48
- isboolean function (builtin), 48
- isfunction function (builtin), 48
- isnative function (builtin), 49
- isnum function (builtin), 49
- isort function (in array), 58
- isstr function (builtin), 49
- JPEG, 98
- JPEG format, 104
- keyboard, 136, 141
- keys function (builtin), 50
- keys function (in ui), 141
- keystroke, 136, 141
- keywords, 161
- labels function (in contacts), 75
- lac, GSM network field, 109
- large function (in ui), 142
- leftkey constant (in ui), 148
- leindex function (in array), 59
- len function (builtin), 50
- len function (in audio), 65
- line function (in graph), 99
- list function (in ui), 142
- literals, 7

- loc, contact field, 71
- Local variables, 11
- log function (in math), 123
- long, GSM network field, 109
- lower function (builtin), 50
- magenta constant (in graph), 88
- math module, 121
- mcc, GSM network field, 109
- md function (autoexec.m), 152
- mem function (in system), 130
- menu command, 136
- menu function (in ui), 143
- menus, 135
- mfont function (in ui), 144
- mkdir function (in files), 82
- MMS, 85
- mnc, GSM network field, 109
- module
  - alias, 35
  - initialization, 35
  - optional, 37
  - prefix, 36
  - version, 37, 38
- modules, 34
- move function (in files), 82
- MP3, 63
- msg function (in ui), 145
- mulaw constant (in audio), 66
- mv function (autoexec.m), 153
- name, contact field, 71
- native object, 49
- native objects, 6
- net function (in gsm), 109
- new function (in array), 60
- new function (in contacts), 76
- new function (in gsm), 110
- note, contact field, 71
- null, 6
  - literal, 10
- num function (builtin), 51
- num function (in time), 133
- Number, 6
- number
  - formatting, 47, 53
  - hexadecimal, 8
  - literal, 7
- number constant (in gsm), 111
- numbers
  - precision, 6
  - range, 6
- open function (in audio), 66
- open function (in io), 115
- operands, 13
- operator
  - arithmetic, 14
  - bitwise, 15
  - boolean, 17
  - comparison, 16
  - concatenation, 16
  - precedence, 14
- optional parameters, 31
- os constant (in system), 131
- own contact, 77

- own function (in contacts), 77
- pager, contact field, 71
- parameter
  - optional, 31
- parameters, 30
- path
  - name, 41
- pcm16 constant (in audio), 66
- pcm8 constant (in audio), 66
- pen, 146
- pen color, 89, 100
- pen function (in graph), 100
- permissions, 165
- pfonts function (in ui), 146
- phone, contact field, 71
- pi constant (in math), 125
- pict, contact field, 71
- play function (in audio), 67
- PNG, 98
- PNG format, 104
- po, contact field, 71
- pointer, 136
- pointing device, 136, 146
- poly function (in graph), 101
- pos function (in audio), 68
- pow function (in math), 123
- precedence, 14
- print, 27
- print function (in io), 116
- println function (in io), 116
- properties file, 161
- ptr function (in ui), 146
- put function (in graph), 101
- query function (in ui), 147
- random function (in math), 123
- raw constant (in array), 63
- raw constant (in io), 112
- rd function (autoexec.m), 153
- read function (in io), 116
- ReadAll, 165
- ReadApp, 165
- ReadDoc, 165
- readln function (in io), 117
- readm function (in io), 117
- receive function (in sms), 127
- record function (in audio), 69
- recording, 63
- rect function (in graph), 103
- recursive function, 11
- red constant (in graph), 88
- region, contact field, 71
- remove function (in array), 61
- ren function (autoexec.m), 153
- rename function (in files), 83
- replace function (builtin), 51
- reserved words, 161
- return, 27, 30
- RGB, 88
- rightkey constant (in ui), 148
- rindex function (builtin), 51
- rindex function (in array), 61
- ring, contact field, 71
- rmdir function (in files), 83

- ro constant (in files), 79
- roots function (in files), 84
- round function (in math), 124
- run function (autoexec.m), 153
- rw constant (in audio), 66
- save function (in graph), 104
- scale function (in graph), 104
- scan function (in files), 84
- seek function (in io), 118
- semicolon, 18
- Send as, 85
- send as, 78
- send function (autoexec.m), 153
- send function (in files), 85
- send function (in sms), 128
- sender, SMS field, 126
- set function (in contacts), 77
- set function (in sms), 128
- set function (in time), 132
- shell, 149
- short, GSM network field, 109
- show function (in graph), 105
- signal function (in gsm), 110
- sin function (in math), 124
- size function (in files), 86
- size function (in graph), 106
- size function (in io), 118
- sleep function (builtin), 52
- SMS control, 155
- sms module, 125
- sort function (in array), 62
- split function (builtin), 52
- sqrt function (in math), 124
- statement list, 18
- statements, 18
- stdin constant (in io), 112
- stdout constant (in io), 112
- stop function (in audio), 69
- str function (builtin), 53
- str function (in time), 133
- stream object, 112
- String, 6
- string
  - literal, 8
- substr function (builtin), 54
- syntax
  - EBNF, 5
  - interactive, 149
- sys constant (in files), 79
- system module, 129
- tan function (in math), 124
- text function (in graph), 107
- text, contact field, 71
- text, SMS field, 126
- time function (in files), 86
- time module, 131
- time, SMS field, 126
- timeout function (in io), 118
- title, contact field, 71
- trim function (builtin), 54
- trunc function (in math), 125
- try
  - module, 37
  - prefix, 37



- type function (autoexec.m), 154
- ui module, 135
- unread, SMS field, 126
- until, 23
- up constant (in graph), 108
- upkey constant (in ui), 148
- upper function (builtin), 54
- url, contact field, 71
- use, 35, 40, 41
- user activity, 140
- utc function (in time), 134
- utf-16be, 162
- utf-16le, 162
- utf-8, 162
- utf16be constant (in io), 113
- utf16le constant (in io), 112
- utf8 constant (in io), 112
- variable, 10
  - global, 11
  - local, 11
- verbosegc function (in system), 130
- version constant (builtin), 55
- video, contact field, 71
- volume function (in audio), 70
- wait function (in audio), 70
- wait function (in io), 119
- wav constant (in audio), 66
- WAV format, 63, 66
- weekofyear function (in time), 134
- while, 22
- white constant (in graph), 88
- write function (in io), 120
- WriteAll, 165
- WriteApp, 165
- WriteDoc, 165
- writeln function (in io), 120
- writem function (in io), 120
- x, pointer event field, 136
- y, pointer event field, 136
- yellow constant (in graph), 88
- zip, contact field, 71