Library

Version 3.00

m Mobile Shell, Library, Version 3.00
Written by Lukas Knecht

www.m-shell.net

Document AB-M-LIB-741

Typeset in Switzerland.

Contents

1. Introduction

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.

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 (Reference, p. 37) 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
```

1.1 Module and Function Availability

The items presented in this manual can be marked by the following:

- A tag with required user permissions (see section A.4 (Reference, p. 82)), for instance
  Permissions: `ReadApp`
  If there is no such tag, no user permissions are required.
- A tag with capabilities required by Symbian platform security (see chapter 6 (Reference, p. 67)), for instance
  Capabilities: `extended`
  If there is no such tag, basic capabilities are sufficient.
- A table describing compatibility issues on specific phones or phones from specific manufacturers, for instance

Compatibility of some function	
ACME phones	Call is ignored

If there is no such table, the function is supposed to work on all phones.

1.2 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 3.1 (Reference, p. 55)):

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

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

Function `files.parse` (p. 32) splits a file name into its four parts:

```
p=files.parse(path)
for k in keys(p) do
  print k,p[k]
end
→ drive c:
   dir \documents\mShell\
   base script
   ext .m
```

To avoid the need for a fully specified file name, each process in **m** maintains a current directory (see `.cd` (p. 7)). 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 double 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\
```

2. Fundamental Modules

2.1 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 1.2 (p. 4).

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. 31), `files.rmdir` (p. 33)

.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. 8)

.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® 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. 8).

.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. 27).

.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. 50).

```
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. 18)

.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. 15)

.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. 17)

.index

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

Searches the string text for the first occurence 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. 16)

.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
```

.isinst

- `function isinst(expression)→ Boolean`

Returns `true` if `expression` is a class instance, `false` if it is any other type.

`isinst(x)` is equivalent to `x is .Instance and x # null`.

```
print isinst(.Instance())
→ true
print isinst(null)
→ false
```

.isinstfunc

- `function isinstfunc(expression)→ Boolean`

Returns `true` if `expression` is an instance function reference, `false` if it is any other type.

```
x:.Instance=.Instance()
print isinstfunc(x.&init)
→ true
print isinstfunc(&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 (Reference, 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 occurences 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")
→ Hellllo 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* occurence 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. 11)

.sleep

- `function sleep(milliseconds)→ null`

Pauses execution for (at least) the number of milliseconds (1/1000 of a second) before returning. If `milliseconds` is negative or zero, execution continues immediately, but other **m** processes immediately get a chance

to run, before they are preempted by the scheduler.

Throws `ExcValueOutOfRange` if `milliseconds` exceeds 2147483 (35 minutes and 47.483 seconds).

```
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[1].

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

Throws `ErrArgument` if `separator` is the empty string.

```
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>,]
```

.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 (Reference, p. 30)):

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

[1] White space: a sequence of characters equal to or less than space. This includes tab and newline.

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 <= start <= len(text)`, or if not `0 <= length <= len(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 upper("Hello")
→ HELLO
print upper("watch out!")
→ WATCH OUT!
```

Constants

- `const` **`version`** `= 3.00`

The current version of **m**. Of course, for a different version this number will be different from 3.00.

2.2 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 `ExcIndexOutOfRange` if not `0 <= start <= len(array)`, or if not `0 <= length <= len(array) - start`, or if any `0 <= indices[i] < len(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 ExcIndexOutOfRange 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. 9).

Throws `ExcIndexOutOfRange` 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. 26)

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 `ExcIndexOutOfRange` 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. 7)

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[2] (one of array.raw, array.fold, array.collate).

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. 26), array.copy (p. 20)

```
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]
```

[2] This sort is always stable.

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. 27), `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
  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. 9).

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. 22)

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` [3] (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. 23)

[3] Sorting is not stable if `mode#raw`.

```
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]
```

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.

2.3 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. 36).

If not absolute, pathes are always relative to the current directory. See also section 1.2 (p. 4).

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. 33)

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. 33)

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 (Reference, p. 78)).

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 1.2 (p. 4).

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.parse

- `function parse(path)→ Array`

Parses a path into its four components and returns them as an array:

Key	Meaning	Type
`drive`	Drive (with trailing colon)	String
`dir`	Directory (with trailing backslash)	String
`base`	Base file name	String
`ext`	Extension (with leading dot)	String

```
path="c:\\documents\\mShell\\script.m";
print files.parse(path)
→ [c:,\documents\mShell\,script,.m]
// Concatenating the four components will
// always produce the original name:
n="";
for p in files.parse(path) do n = n + p end;
print n
→ c:\documents\mShell\script.m
```

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 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 m help files
print files.scan(system.appdir+"*.mhp")
→ [agenda.mhp,app.mhp,array.mhp,audio.mhp,bigint.mhp,
   bt.mhp,cam.mhp,comm.mhp,contacts.mhp,default.mhp,
   files.mhp,graph.mhp,...<28>]
// 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 `files.send`	
Nokia phones before Symbian 8	Call is ignored

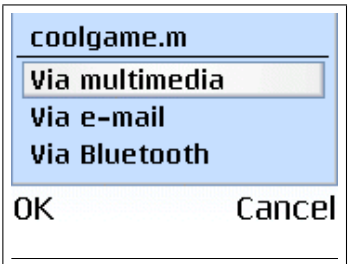
Sends the file `path` over a messaging channel chosen by the user (''Send as''). Channels typically include Bluetooth, MMS, and e-mail. The

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")
```



Series 60 sample screen

UIQ sample screen

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. 50).

2.4 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[4].

```
// 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)
```

[4]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.

There are two special files:

- `const` **`stdin`** `=` *`standard input`* Reads from the console.
- `const` **`stdout`** `=` *`standard output`* Writes to the console[5]

A file always has a *character encoding scheme* (CES) it uses when reading or writing UNICODE® 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 therefore twice the number of characters.

- `const` **`utf16be`** `= 3`

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

- `const` **`bom`** `= 0xfeff`

This is a pseudo-encoding scheme which will determine the real scheme to use depending on the next one to three bytes read. These bytes are analyzed whether they form a BOM (Byte Order Mark) in any given encoding. If there is a BOM, the CES will be set accordingly, and actual reading will start with the data following the BOM. If there is no BOM or the necessary bytes are not available, the CES will be set to *raw*.

To write a BOM to a stream `s` in its current encoding scheme, use the following statement:

[5] `io.stdin` and `io.stdin` represent the same stream; it exists under two different names for historical reasons.

```
if io.ces(s)#io.raw then
  io.write(s, char(io.bom))
end
```

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. 39).

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. 88) 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. 40).

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 (Reference, p. 30). 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. 8)

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[6], 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 airbit AG, www.airbit.ch
io.close(f)
```

io.readm

- function readm(stream,old3rd=false)→ anytype

Reads the next **m** data item from stream, and returns it. The data must have been written using io.writem (p. 46).

[6] end of line is marked by CR-LF, LF, or CR.

Advances the file pointer by the number of bytes read.

The current encoding scheme does not affect how the input data is interpreted.

If `old3rd=true`, data is assumed to be in the (wrong) format written by Symbian 3rd Edition devices with **m** versions prior to 2.01. Under normal circumstances, this parameter is not used.

Throws `ErrEof` if end of file is reached during reading. Throws `ErrCorrupt` if the data in the file is invalid. Throws `ExcNoSuchClass` if the stream contains an instance of a class which is not loaded, i.e. not known to the current process. Throws `ExcUnknownField` if a class has less fields than the instance being read.

Care must be taken when reading class instances which were written with a different class definition. Class fields are simply written and read in order declared when the data was written. Fields added to the class since the data was written are set to `null`.

See `io.writem` (p. 46) 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. 35)

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.

Throws `ExcValueOutOfRange` if `ms` exceeds 2147483 (35 minutes and 47.483 seconds).

The timeout is used in all following reads and writes: whenever an operation does not complete within the given number of milliseconds, it throws `ErrTimedOut`.

```
 // 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, "ErrTimedOut") # 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. 38) returns a value greater than zero),

and returns this stream.

io.wait is most useful when simultaneously processing several input streams (TCP/IP, Bluetooth, IPC), as it avoids the need for a ''busy waiting loop''. See also module async (p. 209).

```
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. 42). `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]];
// and a class instance
class C
  a b
  function init(a, b)
    this.a = a; this.b = b
  end
end
data3=C("String", 24);
f=io.create("sample.dat");
io.writem(f, data1);
io.writem(f, data2);
io.writem(f, data3);
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,[...3]] [One,Two,null,null,V]
print io.readm(f)
→ C(a=String,b=24)
io.readm(f)
→ ErrEof thrown
```

2.5 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 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 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. If `expression` is an array or a class instance, this includes the memory allocated to all elements and fields of the array, recursively.

```
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 **m**.

```
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\| c:\private\a0002f97\| c:\private\e7e0cab7\

The directory where the application files are stored (of the **m** shell, or of the standalone application)

• `const` **`caps`** `= basic | extended | certified | all`
The capabilities granted to this process by the operating system's security platform. Most **m** functions and constants require only `basic` capabilities. The exceptions are marked accordingly. See section 6.1 (Reference, p. 67) for details about capabilities under Symbian OS.

• `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\| c:\Media files\document\mShell\`
The directory where the current **m** script (or executable) is stored.

• `const` **`mdir`** `= c:\system\apps\mShell\| c:\resource\apps\mShell\`
The directory where the **m** resource files are stored.

• `const` **`os`** `= Symbian | Symbian 3rd`
The operating system of the device.

• `const` **`platform`** `= S60 | UIQ`
The (Symbian) platform of the device.

2.6 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 be coarser.

```
print time.get()
→ 63279080895
print str(time.get(), 1, 4)
→ 63279080895.9844
```

See also: .date (p. 9)

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="YMDhmst")→ 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. 52)

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. 51), .date (p. 9)

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
```

2.7 Module zip: ZIP Archives

This module provides read access to ZIP (PKZIP) archive files. Archive members are extracted via ordinary stream objects, to be passed to functions in module io (p. 36).

For instance, the following function extracts all members in a ZIP archive matching a given pattern into the current directory (the default pattern extracts all members). Note that the code does not create any required directories, nor correctly distinguishes between directories and files.

```
function unzip(name, pattern=null)
  z=zip.open(name);
  for f in zip.scan(z, pattern) do
    print "Extracting",f["name"];
    i=zip.extract(z, f["name"]);
    o=io.create(f["name"]);
    b=io.read(i, 256);
    while b#null do
      io.write(o, b); b=io.read(i, 256)
    end;
    io.close(i); io.close(o)
  end;
  zip.close(z)
end
```

zip.close

- `function close(zipfile)→ null`

Closes the ZIP archive `zipfile` previously opened with `zip.open`.

zip.extract

- function extract(zipfile, name)→ Native Object

Opens a stream to extract the member name from the ZIP archive zipfile, and returns it. zipfile must have been previously opened with zip.open.

name is semi-case sensitive: if a member with the same name observing case exists, it is returned, otherwise a member with the same name ignoring case is returned.

Throws ErrNotFound if there is no such member.

The returned stream can be accessed with most functions from module io (p. 36):

- io.read, io.readln, and io.readm read data,
- io.size gets the total number of bytes,
- io.avail gets the number of bytes remaining,
- io.close closes the stream,
- io.ces gets and sets the character encoding scheme. As with files, the default is io.raw.

zip.open

- function open(file)→ Native Object

  Permissions: Read(file)

Opens the ZIP archive with file name file, and returns an object to access it. The object should be closed with zip.close if it is no longer needed.

Throws ErrNotFound if there is no such archive, and ErrCorrupt if the archive is not valid.

zip.scan

- function scan(zipfile,name=null)→ Array

Scans the ZIP archive zipfile for all members matching name. name is

not case sensitive and can contain the wildcards * and ?. If name=null, all members are returned.

Returns an array with one element for each member found, each element being an array with the following keys:

Key	Meaning	Type
name	Member name	String
size	Uncompressed size	Integer
csize	Compressed size	Integer
crc	CRC-32 checksum	Integer

```
z=zip.open('ZipTest.zip')
for f in zip.scan(z, '*AudioTest.*') do
  print f
end
→ [tests\AudioTest.mid,1983,510,2487703623]
  [tests\AudioTest.mm,2416,952,1954653783]
```

3. User Interface

3.1 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. 66).

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. 79) 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. 78) function.

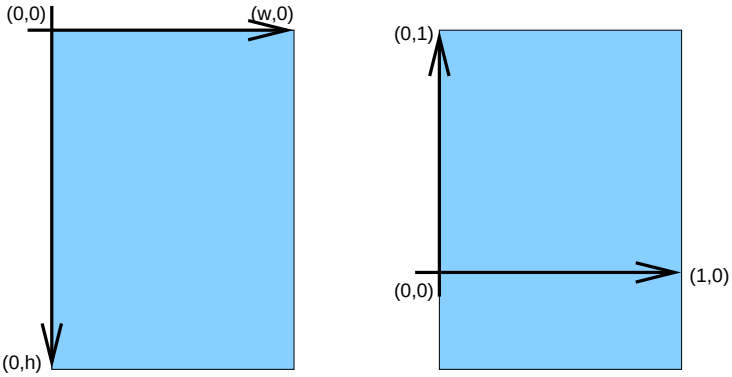
Graphic objects are drawn on the canvas by calling the corresponding functions. The canvas is not transferred to screen until `graph.show` (p. 78) is called, or the operating system requests redrawing.

Coordinates

Position and size of graphic objects are given by coordinates. This module supports two modes for specifying coordinates (see also `graph.scale` (p. 77)):

- *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)

Drawing coordinates are always relative to the current clipping rectangle. See also `graph.clip` (p. 63).

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** = 0xffffff
- 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. 61)), 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. 73).
- 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. 62).

Alpha Blending

All drawing operations can optionally be modified such that the item drawn is blended with the background. Blending is configured by a parameter α, a number between 0 and 1 indicating to extent the background image is blended in:

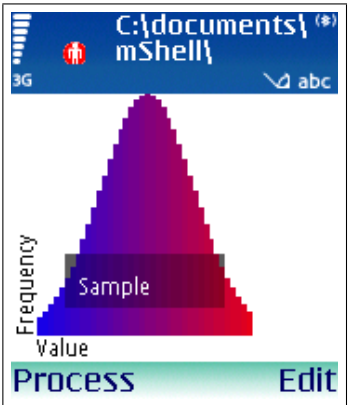
- $\alpha = 0$: no blending, the background is ignored. This is the default setting.
- $0 < \alpha < 1$: the background is blended in.
- $\alpha = 1$: the background is completely blended in, i.e. the drawing operation has no effect.

α is set via graph.alpha (p. 61). Setting it to a value other than 0 causes a small performance penalty on drawing operations.

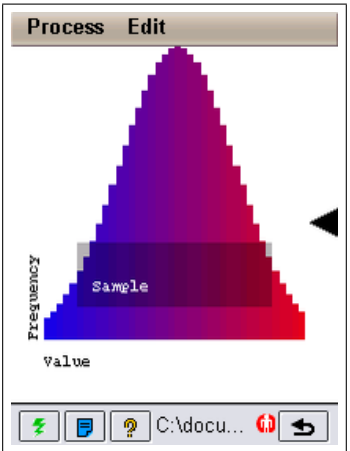
Simple Example

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, then blends a black rectangle over it with the text ''sample''.

```
// 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.brush(graph.black);
graph.alpha(0.7);
graph.rect(0.2, 0.2, 0.6, 0.2);
graph.pen(graph.white);
graph.alpha(0);
graph.text(0.25, 0.25, "Sample");
graph.show();
```



Series 60 sample screen



UIQ sample screen

graph.alpha

- `function alpha(factor)→ Number`
- `function alpha()→ Number`

Gets or sets the background blending factor α applied to all drawing operations. With one argument, sets the blending factor, and returns the old factor. Without arguments, returns the current factor.

α is a value between 0 and 1, inclusive. See section iwref-sec:alphablending for more information.

```
// blend all drawing operations with 30%
// of the background
graph.alpha(0.3)
```

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 3.1 (p. 58) 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 3.1 (p. 58) for the definition of colors.

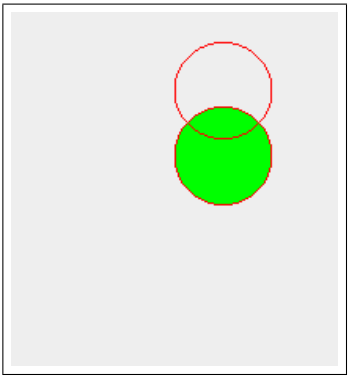
```
// fill the objects with white
graph.brush(graph.white)
```

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`

Fills the entire view with the current background color, as defined by `graph.bg` (p. 61).

graph.clip

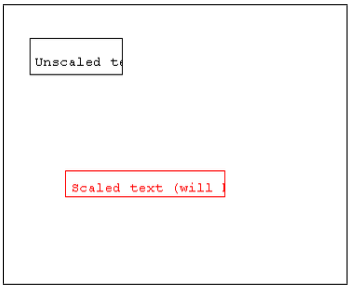
- `function clip()→ Array | null`
- `function clip(xywh)→ Array | null`

Returns the current clipping rectangle, or `null` if there is no clipping (default). Without arguments, the clipping rectangle is not modified.

With one argument `xywh=[x,y,w,h]`, the clipping rectangle is set to this rectangle (scaled if in scaled mode). All subsequent drawing operations are relative to the upper left (unscaled) or lower left (scaled) corner of the clipping rectangle.

With one argument `xywh=null`, the clipping rectangle is removed, and the drawing origin is set back to the upper left (unscaled) or lower left (scaled) corner of the canvas.

```
// unscaled clipping
graph.scale(false); graph.pen(graph.black);
graph.rect(20, 30, 100, 40);
graph.clip([20, 30, 100, 40]);
graph.text(5, 30, "Unscaled text (will be clipped)");
// scaled clipping
graph.scale(true); graph.pen(graph.red);
print graph.clip(null);
→ [0.0694444444,0.8958333333,0.3472222222,
   0.1388888889]
graph.rect(0.2, 0.3, 0.6, 0.1);
graph.clip([0.2, 0.3, 0.6, 0.1]);
graph.text(0.02, 0.02, "Scaled text (will be clipped)");
```



Sample **m** *screen*

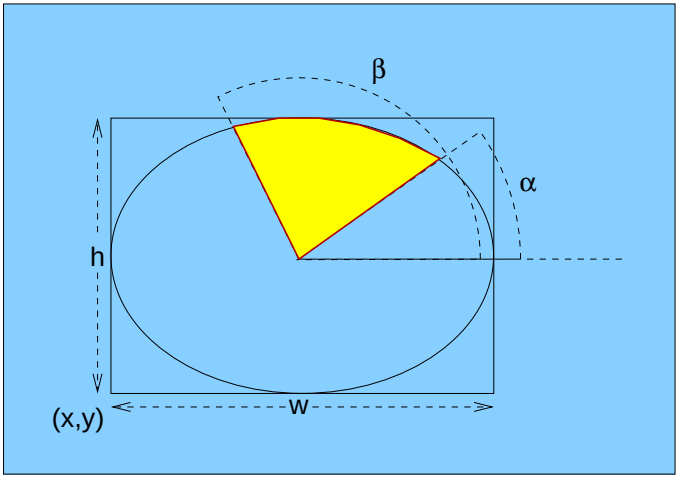
graph.ellipse

- `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:

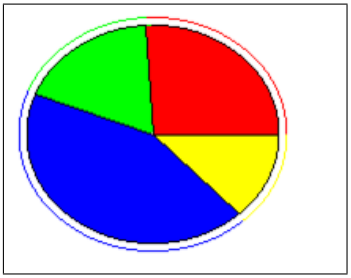
- With four arguments, draws an ellipse into the rectangle with corner at `x,y`, width `w` and height `h`. The outline is drawn with the current pen color, and the ellipse is filled with the current brush color.
- With six arguments and the brush enabled, draws the outline of a pie slice with the current 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 parellel arc (brush disbled)
  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. 81) 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. 92) for the definition of fonts, and `graph.text` (p. 81) for an example using fonts.

graph.full

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

Compatibility of function `graph.full`	
Sony Ericsson phones[a].	Restricted menu access

[a] 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. 79).

The following function fills the screen (not the canvas) with an ellipse in

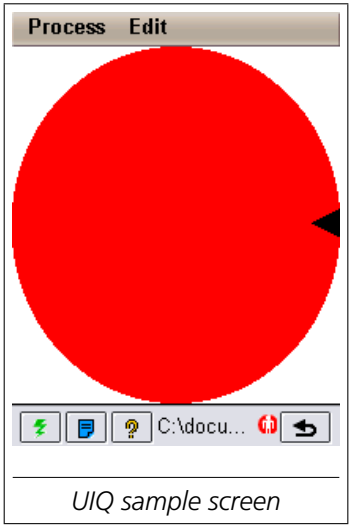
a given color:

```
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()
```

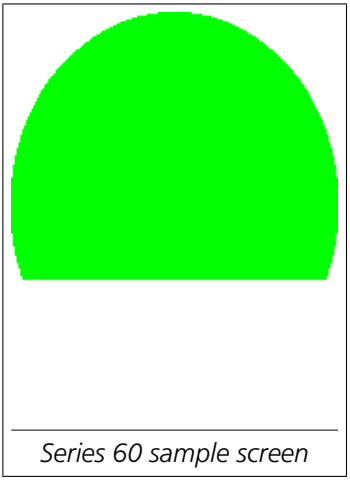
Series 60 sample screen



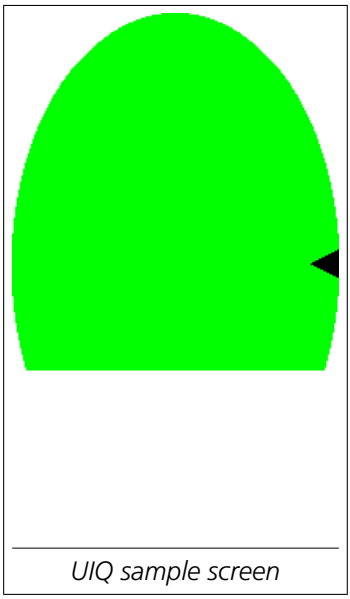
UIQ sample screen

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()
```



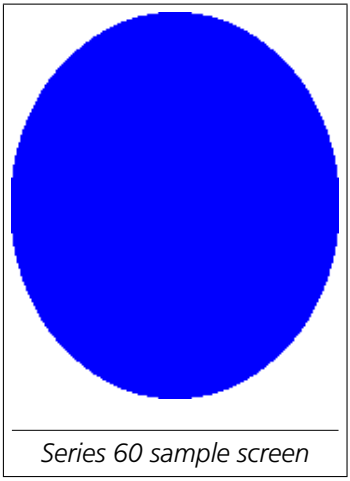
Series 60 sample screen



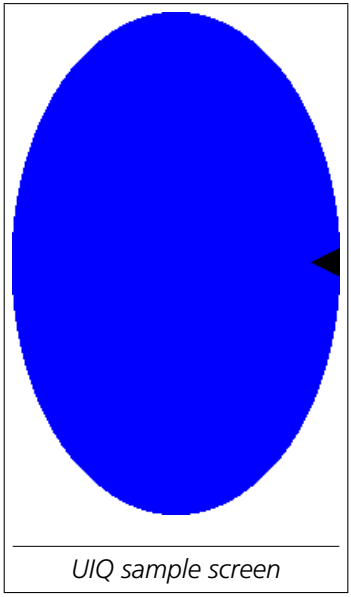
UIQ sample screen

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()
```



Series 60 sample screen



UIQ sample screen

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 3.1 (p. 58)).

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. 74).

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. 74).

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. 70), 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. 79) 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()
```

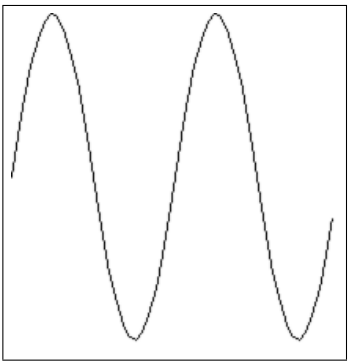
Sample **m** *screen*

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 3.1 (p. 58) for the definition of colors.

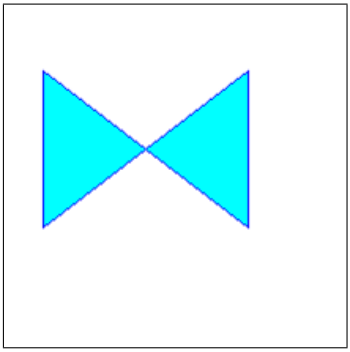
```
// use a slightly dark magenta pen
graph.pen(0xc000c0)
```

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 <= i < len(x) - 1`), with the last (closing) line going from `(x[len(x)-1],y[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+len(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.

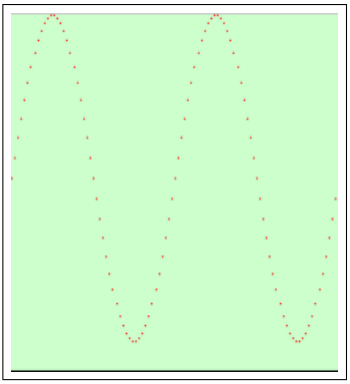
If the third parameter is an icon, draws icon with upper left corner (x,y). If the icon has a mask, only opaque pixels are drawn.

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()
```

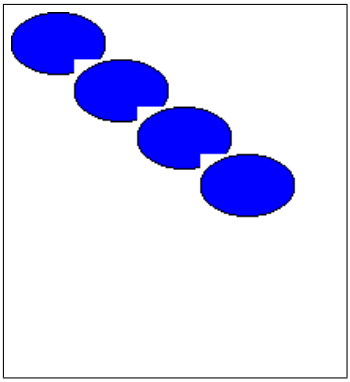


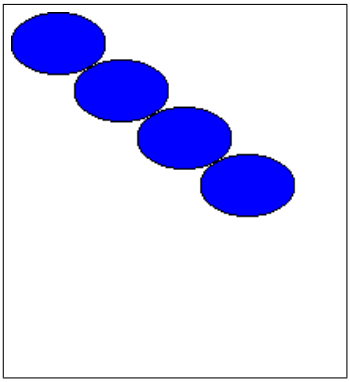
Sample **m** *screen*

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

background:

```
// 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 suffices 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 3.1 (p. 57).

graph.screen

- `function screen()→ Native Object`
- `function screen(x, y, w, h)→ Native Object`

  Permissions: `ReadApp`

Produces an icon of the image (or a portion of it) on the device screen and returns it. If **m** is running in the background, this takes a device screen shot.

Without arguments, the icon will contain the entire screen image. With four arguments, only the rectangle between screen coordinates `(x,y)` and `(x+w,y+h)` will be copied. In scaled mode, `x`, `y`, `w` and `h` are scaled.

```
// make the canvas as large as the screen
graph.size(graph.full(true));
// take a screen shot and save it to screen.gif
i=graph.screen();
graph.put(0,0,i);
graph.save("screen.gif");
```

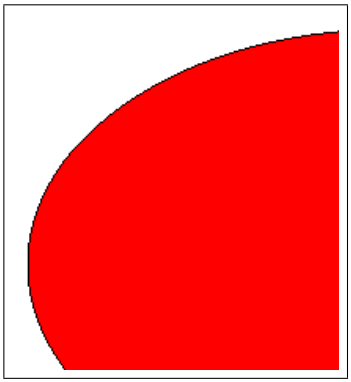
graph.show

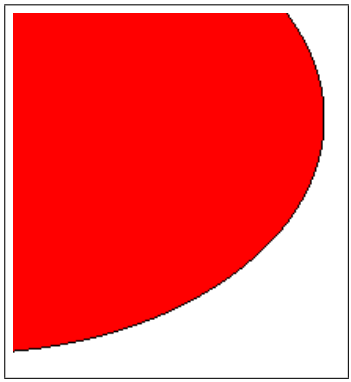
- `function show()→ null`
- `function show(x, y)→ null`

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 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, bounds=false)→ Array
- function size(wh)→ 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 `icon` argument, returns the size (width and height) of the icon.

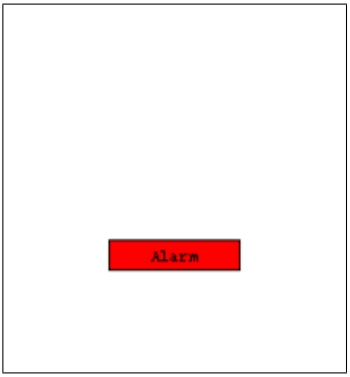
With one `text` argument and an optional boolean argument, returns returns the size (width and height) of the text if it were drawn using the current font. If `bounds=true`, the size of the bounding box around the text is returned, considering the actual ascents and descents in the text. If `bounds=false`, the height returned only depends on the font and can thus be used to vertically align chunks of text.

With one array argument `wh=[w,h]` or 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. 78) 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

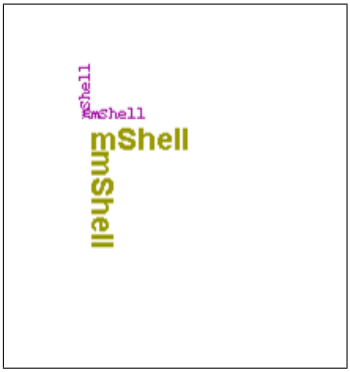
- `function text(x, y, text, direction=0)→ null`

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*

3.2 Module `ui`: User Interface Functions

This module provides functions to display standard dialogs and menus and to modify the **m** user interface.

`ui.busy`

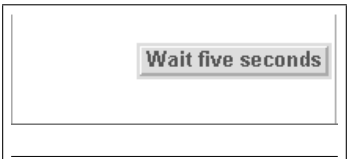
- `function busy(activity)→ null`
- `function busy()→ null`

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
```

Series 60 sample screen



UIQ sample screen

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® number, and can thus be converted with `.char` (p. 8). Codes for navigation and system keys are device specific. Some important keys are defined as constants (see 3.2 (p. 97)).

  `ui.keys` (p. 88) 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. 91) 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. 95) 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. Throws `ExcValueOutOfRange` if `ms` exceeds 2147483 (35 minutes and 47.483 seconds).

Keyboard, menu and pointer can all be monitored together in a single `ui.cmd` call.

See `ui.keys` (p. 88) for an example using the keyboard, `ui.menu` (p. 91) for an example using menus, `ui.ptr` (p. 95) for an example using the pointer.

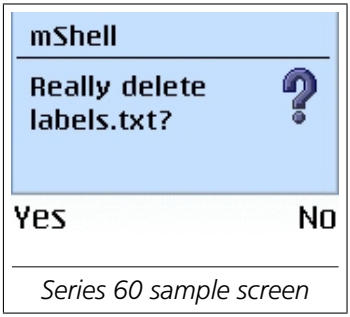
ui.confirm

- `function confirm(question, title="mShell")→ Boolean`

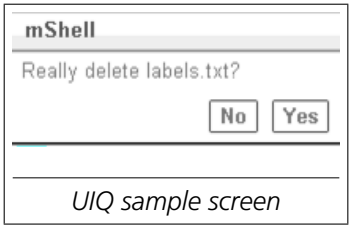
Shows a simple dialog displaying `question` in a dialog with title `title`. The dialog asks the user for confirmation, presenting two buttons or soft keys with the options ''yes'' and ''no''.

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
```



Series 60 sample screen



UIQ sample screen

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)
```

Series 60 sample screen

UIQ sample screen

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`

Compatibility of function `ui.form`
Nokia phones silently ignore the `title` parameter.

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:

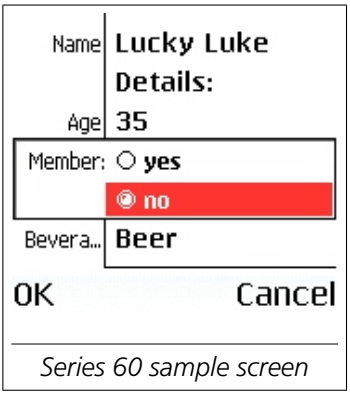
Data Type	Field Type
String without `\n`	Single line text editor
String with `\n`	Multi-line text editor
String with trailing `ui.secret`	Password editor indexsecret editor
Number	Number editor (floating point)
Boolean	Check box or popup yes/no choice
Array	Combo box or popup multiple choice

For the multi-line and secret editors, a terminating `\n` or `ui.secret` will be removed, so an empty multi-line field is defined by a single newline character, and an empty secret field by `ui.secret`.

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.

```
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]
```

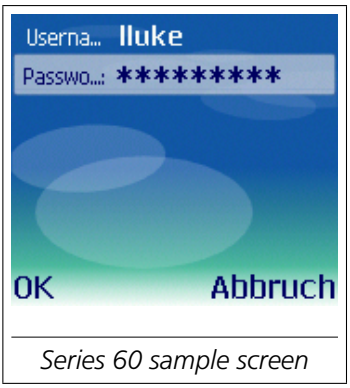


Series 60 sample screen

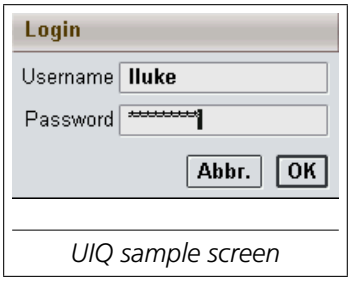
UIQ sample screen

A typical username/password dialog is obtained as follows:

```
old=["Username":"", "Password":ui.secret];
new=ui.form(old, "Login");
print new
→ [lluke,rosinante]
```



Series 60 sample screen



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
  elsif 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. 83). 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 ui.large	
Sony Ericsson phones	UI size change is not possible; function always returns false.

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`:

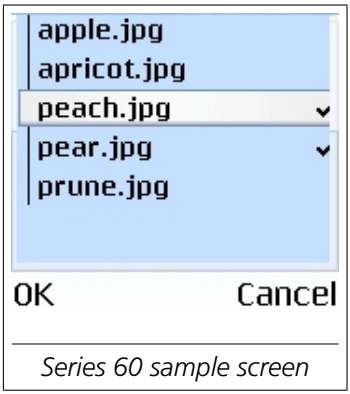
- If `multiple=false`, only one item can be selected. This is usually simply the highlighted (current) item.
- If `multiple=true`, multiple items can be selected. These are usually the marked items.

Initially, the items indexed in `init` will be selected (or marked).

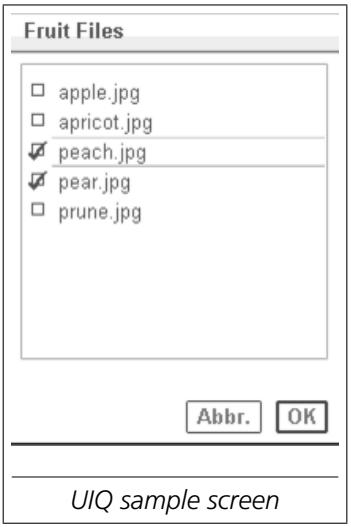
If the user presses ''ok'', this function returns the indices of the items selected by the user, i.e. an array of numbers indexing into `items`. If the user presses ''cancel'', `null` is returned.

`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]
```



Series 60 sample screen



UIQ sample screen

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. 83).

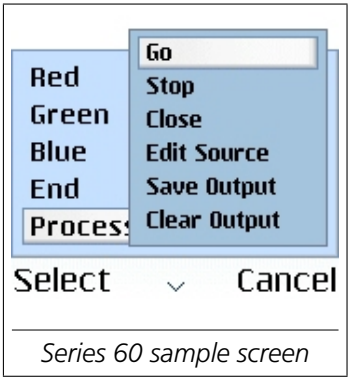
If keepold=true, the standard process menu will be added at the end, as a submenu. If keepold=false, the standard functions are not available, preventing the user from easily stopping or closing the running process.

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. 83).

```
ui.menu("Colors", ["Red", "Green", "Blue", "End"]);
while true do
  c=ui.cmd();
  if c="End" then break end;
  print c,"chosen"
end
```



Series 60 sample screen

UIQ sample screen

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.mode

- function mode()→ Number
- function mode(newmode)→ Number

Compatibility of function ui.mode	
Symbian 2nd Edition or Sony Ericsson phones	Only support mode 0.

Gets or sets the screen / user interface orientation mode.

Without an argument, returns the current mode. With a single argument, sets the mode to newmode and updates the user interface accordingly.

The following modes are available:

Value	Description
0	Unspecified: the screen mode of **m** follows the orientation implied by the device (e.g. flip open or closed).
1	Portrait: **m** is always shown in portrait mode.
2	Landscape: **m** is always shown in landscape mode.

Throws ExcValueOutOfRange if the desired mode is not supported.

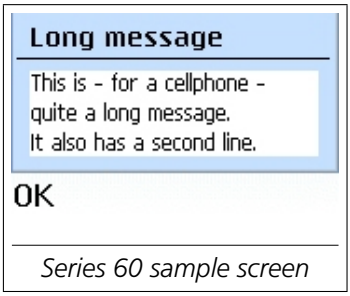
```
// force mode to landscape
ui.mode(2)
```

ui.msg

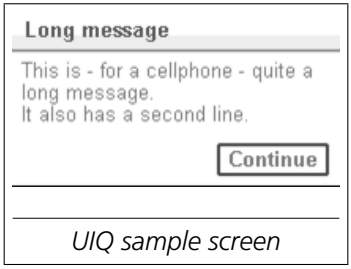
- function msg(message, title="mShell")→ null

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");
```



Series 60 sample screen



UIQ sample screen

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. 83). 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[1].

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).

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
```

[1] mVNC has limited support for the Series 60 pointer via the mouse.

ui.query

- `function query(prompt, title="mShell", value="")→ String|Number|null`

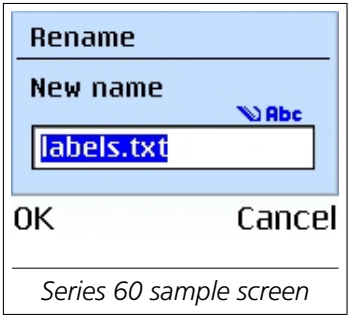
Displays a dialog querying for a single text input. The input field is initialized with `value`, and labelled with `prompt`.

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.

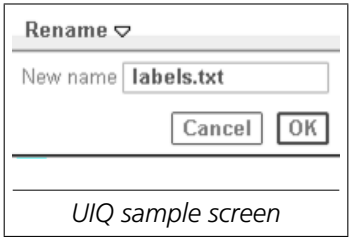
If the user presses ''ok'', this function returns the value entered by the user. If the user presses ''cancel'', `null` is returned.

The same effect can be achieved with `ui.form` (p. 86), 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
```



Series 60 sample screen



UIQ sample screen

ui.save

- `function save(path)→ null`

  Permissions: `Write(path)`

Saves the current contents of the console to file `path`. This has the same

effect as manually executing **Process→Save Output**, except that path is relative to the current directory of the process.

The text is written using the current source file encoding.

```
print "Hello world";
→ Hello world
ui.save("output.txt");
print io.readln(io.open("output.txt", false, io.bom))
→ Hello world
```

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 **downkey2** = *other down key code* On UIQ devices, the four way ''down'' navigation key; on S60 devices, same as ui.downkey.
- 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 **secret** = "\u0001" The secret input field mark.
- const **upkey** = *up key code* The ''up'' navigation key.
- const **upkey2** = *other up key code* On UIQ devices, the four way ''up'' navigation key; on S60 devices, same as ui.upkey.

3.3 Module vibra: Vibration Control

Compatibility of module vibra	
Nokia phones before Symbian 8[a]	Internal Error

[a] See also Forum Nokia, Developer Platform 2.0: Known Issues, 5.13

This module provides simple functions to control the device vibration

feature of some devices.

vibra.off

- `function off()→ null`

Turns the vibration off. If the device is not vibrating, the call is ignored.

vibra.on

- `function on(duration=0)→ null`

Turns the vibration on for the specified duration (in milliseconds). If `duration=0`, vibration is turned on until `vibra.off` (p. 98) is called.

This function returns immediately, before the specified time has passed.

Throws `ExcValueOutOfRange` if the duration is outside the valid range (0 to 65535).

```
// vibrate for one second:
vibra.on(1000)
// another way to vibrate for one second:
vibra.on();
sleep(1000);
vibra.off()
```

4. Mathematics

4.1 Module `bigint`: Arbitrarily Large Integers

This module supports calculations with big integers. The maximum (or minimum) value for a big integer is limited only by available memory. All calculations are performed with full precision.

Big integers are native objects. Three functions convert between big integers and other representations:

- `bigint.new` (p. 102) creates a new big integer from a number, a string (in a given base, e.g. hexadecimal), or another big integer.
- `bigint.num` (p. 102) converts a big integer to a number (potentially loosing significant digits).
- `bigint.str` (p. 103) converts a big integer to a string encoded in a given base.

The big integer arguments of all functions can also be specified as a number or as a string encoding a decimal number:

```
a=bigint.mul("33333333333333333333333333333333333", -2);
print a, bigint.str(a)
→ bigint@414ffc -66666666666666666666666666666666666
```

`bigint.abs`

- `function abs(p)→ Native Object`

Computes the absolute value of `p` as a big integer.

```
r=bigint.abs("-3141592653589793238462643");
print bigint.str(r)
→ 31415926535897932384626433
```

bigint.add

- `function add(p, q)→ Native Object`

Computes the sum of `p` and `q` as a big integer.

```
r=bigint.add("12345678901234567890123456789",
             8765432110);
print bigint.str(r)
→ 12345678901234567891000000000
```

bigint.cmp

- `function cmp(p, q)→ Number`

Compares `p` and `q`:

- Returns `-1` if `p < q`.
- Returns `0` if `p = q`.
- Returns `1` if `p > q`.

```
p=bigint.new("100000000", 16);
q=bigint.new(4294967296);
print bigint.cmp(p, q)
→ 0
```

bigint.div

- `function div(p, q)→ Native Object`

Computes the quotient of `p` and `q` as a big integer. Throws `ErrDivideByZero` if `q=0`.

```
r=bigint.div("12345678901234567890123456789O",
             1234567890);
print bigint.str(r)
→ 100000000010000000001
```

bigint.mod

- function mod(p, q)→ Native Object

Computes the remainder of p and q as a big integer. Throws ErrDivideByZero if q=0.

```
r=bigint.mod("12345678901234567890123456789З",
             1234567890);
print bigint.str(r)
→ 3
```

bigint.mul

- function mul(p, q)→ Native Object

Computes the product of p and q as a big integer.

```
p=bigint.new(333333333333333);
r=bigint.mul(p, p);
print bigint.str(r)
→ 111111111111110888888888888889
```

bigint.neg

- function neg(p)→ Native Object

Computes the value of p with sign changed.

```
r=bigint.neg("3141592653589793238462б4")
print bigint.str(r)
→ -31415926535897932384626б4
```

bigint.new

- `function new(p)→ Native Object`
- `function new(string, base=10)→ Native Object`

Creates a new big integer with the value of `p`. `p` can be:

- Another big integer. In this case a copy of `p` is returned.
- A number. Digits after the decimal points are ignored, and for values outside the range -2^{63} to $+2^{63}-1$, the result is undefined.
- A string encoding an integer in the given base. Valid bases are in the range 2 (binary) and 36 (using letters `A` to `Z` and `a` to `z` for digits `11` to `36`).

  Leading and trailing blanks in the string are ignored.

  Throws `ErrArgument` if the base is out of range or the string contains invalid characters.

```
m=bigint.new(-18513.7);
print bigint.str(m)
→ -18513
m=bigint.new("ffffffffffffffff", 16);
print bigint.str(m, 4)
→ 3333333333333333333333333333333
```

bigint.num

- `function num(p)→ Number`

Converts the big integer `p` to a number. If `p` is outside the range -2^{63} to $+2^{63}-1$, the result is undefined.

```
r=bigint.div("12345678901234567890", 1234567890);
print bigint.num(r) / 2
→ 5000000000.5
```

bigint.pow

- function pow(p, q)→ Native Object
- function pow(p, q, m)→ Native Object

Efficiently computes p^q as a big integer. With three arguments, computes the remainder of dividing p^q by m.

Throws ErrArgument if q<0. Throws ErrDivideByZero if m=0.

```
// perform RSA encryption with a 256-bit key
e=bigint.new("771558090212905276225534849558673251628 5"+
             "075433134084976912888193193008984746 7");
m=bigint.new("115733713531935791433830227433800987744 9"+
             "565246692445521247590128659296812307 09");
c=bigint.pow("369519557033938821820522315342888319207 3"+
             "329889262155589752278898769206369823",
              e, m);
print bigint.str(c)
→ 20909637262569569616272545802765117583929326309338 05
  17450963329807056506783 28
```

bigint.str

- function str(p, base=10)→ String

Converts the big integer p to a string in the given base. Valid bases are in the range 2 (binary) and 36 (using letters a to z for digits 11 to 36).

```
// convert a large decimal to a large hexadecimal number
s=bigint.str("12345678901234567890123456789 0", 16);
print s
→ 18ee90ff6c373e0ee4e3f0ad2
```

bigint.sub

- function sub(p, q)→ Native Object

Computes the difference of p and q as a big integer.

```
r=bigint.sub("12345678901234567890123456789O",
             -8765432110);
print bigint.str(r)
→ 123456789012345678910000000000
```

4.2 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 uniformely 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` π.

5. Personal Data

5.1 Module agenda: Agenda Database

This module allows to read and manipulate the agenda (calendar and to-do list) stored on the phone. There are different types of agenda entries, each type identified by its flag:

- Appointment (`agenda.appt` flag): an entry starting at a date and time and ending on the same day, e.g. a team meeting.
- Event (`agenda.event` flag): an entry starting at a date and ending on a date, e.g. holidays.
- Anniversary (`agenda.anniv` flag): an entry occuring at a date, with an optional base year (e.g. the year of birth).
- To-do list item (`agenda.todo` flag): an entry with a due date and a priority. When done, it also gets a done (''crossed out'') date.

The standard calendar application on the phone often does not support all entry types and attributes.

In the phone's database, an agenda entry is identified by its id, an integer number.

Agenda Fields

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

Key	Meaning	Type	Used in			
			appt	event	anniv	todo
alarm	Alarm date/time	Seconds	×	×	×	×
base	Base year	Integer			×	
done	Done date	Seconds				×
end	End date/time	Seconds	×	×		×
flags	Entry flags (see below)	Integer	×	×	×	×
loc	Location	String	×	×	×	×
prio	Priority	Integer				×
rep	Repeat details (see below)	Array	×	×	×	×
start	Start date/time	Seconds	×	×	×	×
text	Entry text	String	×	×	×	×

Key names are not case sensitive.

All dates and times of an entry are represented as seconds since the start of year zero in local time (see also module time (p. 50)). Valid dates are January 1st, 1980 or 1900 to December 31st, 2100. The functions of this module throw ExcValueOutOfRange if a date outside this range is used. The only exception is the base year (base) of an anniversary entry, which is simply an integer indicating any year.

The order of fields in the array describing an entry is arbitrary. Arrays returned by functions in this module always start with the two fields text and flags.

Agenda Entry Flags

The flags field is a bitwise combination of the following values:

- const **anniv** = 4 Entry is an anniversary.
- const **appt** = 1 Entry is an appointment.
- const **done** = 32 To-do entry is done.
- const **event** = 2 Entry is an event.
- const **rep** = 16 Entry is repeated.
- const **remind** = 64 Entry is a reminder.
- const **todo** = 8 Entry is a to-do list item.

Flags can be used to select entries in agenda.find (p. 114), and they must be used to indicate the type of the new entry in agenda.add (p. 113).

For use in agenda.find (p. 114), there is also the value

- const **all** = 127 All flags combined.

Repetitive Entries

All dated entries can be repetitive: a repetitive entry is automatically repeated according to its repeat details. For instance, an anniversary is typically repeated on the same date every year. Repeating an entry does not duplicate the entry; deleting or updating a repetitive entry also deletes or updates all its repetitions.

In **m**, the repeat details of an entry are represented as an array stored in the entry's rep field. **m** recognizes the following keys of this array, with the corresponding data type:

Key	Meaning	Type
end	Repeat end date	Seconds
interval	Repeat interval (days, months, years)	Integer
type	Repeat type (see below)	Integer
when	Repeat selection (see below)	Array of Integer

If end=null (the default), the entry is repeated forever. The default interval is 1. type must be one of the following six values:

- const **daily** = *Repeat daily.*

Repeat the entry every interval days.

```
// plan for an 30 minute exercise at 8am
// every three days, starting today
today=86400 * math.trunc(time.get() / 86400);
e=["text":"exercise",
   "start":today+8*3600,
   "end":today+8*3600+1800,
   "flags":agenda.appt,
   "rep":["type":agenda.daily, "interval":3]]
```

- const **weekly** = *Repeat weekly.*

Repeat the entry every interval weeks, on the week days indicated by

`when`. Week days start with zero as Monday; see also `time.dayofweek` (p. 50).

```
// repeat every week on Tuesday and Friday
e["rep"]=["type":agenda.weekly, "when":[1,4]]
```

- `const` **`monthlydate`** = *`Repeat monthly, at given dates.`*

Repeat the entry every `interval` months, on the days indicated by `when`.

```
// repeat every two months on the 10th and 25th
e["rep"]=["type":agenda.monthlydate,
          "interval":2, "when":[10,25]]
```

- `const` **`monthlyday`** = *`Repeat monthly, at given days of weeks.`*

Repeat the entry every `interval` months, on the week days in the weeks indicated by `when`: `when[2*i]` indicates the week of the month (1 is the first, 4 is the fourth, 5 the last), and `when[2*i+1]` indicates the day of week (0 is Monday).

```
// repeat every month on the Tuesday (1) of the 2nd
// week (2), and on the Tuesday (1) of the last week (5)
e["rep"]=["type":agenda.monthlyday, "when":[2,1,5,1]]
```

- `const` **`yearlydate`** = *`Repeat yearly, at a given date.`*

Repeat the entry every `interval` years, on the date implied by the entry's start date. This repeat type is typically used for anniversaries.

```
// repeat every year
e["rep"]=["type":agenda.yearlydate]
```

- `const` **`yearlyday`** = *`Repeat yearly, at a given day of a week of a month.`*

Repeat the entry every `interval` years, on the day indicated by `when`: `when[0]` indicates the month, `when[1]` the week of the month (1 ist the first, 4 is the fourth, 5 is the last), and `when[2]` the day of week (0 is Monday).

```
// repeat yearly, on Sunday (6) of the 1st week in April
e["rep"]=["type":agenda.yearlyday,"when":[4,1,6]]
```

agenda.add

- function add(entry)→ Number

  Permissions: WriteApp

Add an entry to the agenda database, and return its id. The entry must be an array with keys from the above tables. The entry type is derived from the flags array element; if there is no flags element, an agenda.appt entry is added.

```
// Add a 30 minute meeting starting in two hours,
// in the CEO's office
start=time.get()+2*3600;
e=["text":"Group meeting",
   "flags":agenda.appt,
   "start":start,
   "end":start+1800,
   "loc":"CEO's office"];
agenda.add(e)
→ 402653204
// Add an anniversary, repeating every year
e=["text": "Shakespeare's Birthday",
   "flags": agenda.anniv,
   "start": time.num("2005-04-23"),
   "base": 1564,
   "rep": ["type":agenda.yearlydate]];
agenda.add(e)
→ 117440532
```

agenda.delete

- function delete(id)→ null

  Permissions: WriteApp

Delete the contact with the given id.

Throws ErrNotFound if there is no such contact.

```
// delete the anniversary added in the add example
agenda.delete(117440532)
```

agenda.find

- `function find(start=null, end=null, flags=agenda.appt | agenda.event | agenda.anniv | agenda.rep)→ Array`

Permissions: `ReadApp`

Searches the agenda for entries overlapping with the period between `start` and `end`, and with an entry type indicated by `flags`. The default flags exclude to-do list entries.

Repeated entries are only reported if the first repetition falls within the period. Use `agenda.findall` (p. 115) to find all events within a period, including repetitions.

`start` and `end` must be given in seconds since year zero; `start=null` indicates the earliest possible start date, `end=null` the latest possible end date.

```
// get the number of entries in the agenda
print len(agenda.find(null, null, agenda.all))
→ 53
// print the text and start of today's entries
today=86400*math.trunc(time.get()/86400);
for id in agenda.find(today,today+86400) do
  e=agenda.get(id);
  print e["text"], time.str(e["start"], "hh:mm")
end
→ ...
   Group meeting 18:40
   ...
// delete all entries up to now, excluding repetitives
for id in agenda.find(null, time.get(),
                      agenda.all & ~agenda.rep)
  agenda.delete(id)
end
```

agenda.findall

- `function findall(start, end, flags=agenda.appt | agenda.event | agenda.anniv | agenda.rep)→ Array`

  Permissions: `ReadApp`

This is similar to `agenda.find` (p. 114), except that all instances of repeated entries are reported if they fall within the period. The function returns an array with an element for each instance found. Each element is again an array with the following elements:

Key	Meaning
`id`	The id of the entry, as returned by `agenda.find`.
`at`	The start time of the instance. For a non-repeated entry, this is the same as the entry's start time. For a repeated entry, this is the start time of the instance falling within the selected period.

```
// print the text and start of today's instances
today=86400*math.trunc(time.get()/86400);
for idtime in agenda.findall(today,today+86400) do
  e=agenda.get(idtime["id"]);
  print e["text"], time.str(idtime["at"], "hh:mm")
end
→ ...
   Group meeting 18:40
   Weekly Yoga session 20:30
   ...
```

agenda.get

- `function get(id)→ Array`

  Permissions: `ReadApp`

Get the fields of the agenda entry with id `id`.

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

```
// get the entry added before
e=agenda.get(402653204);
print e
→ [Group meeting,1,63284611200,63284613000,
   CEO's office]
print time.str(e["start"])
→ 2005-05-17 18:40:00
```

agenda.set

- `function set(id, entry)→ null`

  Permissions: `WriteApp`

Updates the entry with id `id`, updating the fields in array `entry`. `entry` must be an array with keys from the above tables. Fields which are `null` in the array are cleared in the entry.

```
// Change the location of the group meeting
agenda.set(402653204, ["loc":"My office"])
// Set all done entries in the to-do list to "not done"
ids=agenda.find(null, null, agenda.todo | agenda.done);
for id in ids do
  agenda.set(id, ["done":null])
end
```

5.2 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
adr	Address (street)
birth	Birthday
cell	Cellphone number
company	Company name
country	Country
email	e-mail address
extadr	Additional address
extname	Additional name
fax	Fax number
fname	First name
loc	Locality (city)
name	(Family) name

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

Suffix	Meaning
.home	Home address or phone
.work	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. 50).
- 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
```

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` 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. 50).

```
// 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[1] 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
```

[1] A standard defined by the Internet Mail Consortium, see www.imc.org/pdi/vcardoverview.html.

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.

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
```

6. Communications

6.1 Module `bt`: Bluetooth Communication

This module provides access to Bluetooth® wireless communication with other Bluetooth equipped devices. The supported functions are:

- Obtaining the own bluetooth address and name, and modifying the latter.
- Getting and setting the Bluetooth visibility flag.
- Scanning for visible devices and obtaining the address, name and class, also interactively.
- Creation of services (passive connections), either directly using a channel number, or by registering with an UUID for service discovery.
- Connecting to services (active connections), either directly using a channel number, or by looking an UUID up via service discovery.

Terminology

Bluetooth is a relatively complex technology. The following is a quick crash course of the key concepts required to completely understand this module. For more information and detailed specifications, see www.bluetooth.org.

- **Device Address:** Each Bluetooth device is identified by a unique 48 bit address. In this module, an address is a string of six hexadecimal bytes, separated by colons, e.g. `"00:E0:03:5E:AF:CD"`, or `"0:e0:3:5e:af:cd"`.

- **Device Name:** Each Bluetooth device can have a freely assignable name. A well chosen name helps in distinguishing visible devices, but is of little use when trying to automatically identify or find a device.

- **Device Class:** Each Bluetooth device has a class defining its type and capabilities. The device class is a 24 bit integer, encoded as follows:

Bits	Value	Contents
0-1		Always zero
2-7		**Minor device class:** interpretation depends on Major device class
8-12		**Major device class:**
	0	Miscellaneous
	1	Computer
	2	Phone
	3	LAN/Network access point
	4	Audio/Video
	5	Peripheral (mouse, joystick, keyboard)
	6	Imaging (printer, display, scanner, camera)
	7	Wearable
	31	Uncategorized
13-23		**Service class:**
16	1	Positioning (GPS)
17	1	Networking (LAN)
18	1	Rendering (Video and Audio)
19	1	Capturing (Video and Audio)
20	1	Object Transfer (vCal, vCard)
21	1	Audio
22	1	Telephony
23	1	Information (WWW/WAP-Servers)

- **SDP (Service Discovery Protocol):** A mechanism to advertise services (e.g. data synchronization, printing, scanning, or own services), and discover them. Services are identified by UUIDs.

- **UUID (Universally Unique Identifier):** This is a 128 bit (16 byte) quantity. In Bluetooth, each service has one or more UUIDs

assigned: when creating a service, a UUID should be assigned to it (see bt.start (p. 133)).

In this module, a UUID is represented as an array of four nonnegative numbers, starting with bits 127 to 96, and ending with bits 31 to 0. See also bt.uuid (p. 135).

In Bluetooth, often only 32 bits of the UUID are specified. Such an UUID maps to a 128 bit UUID by adding fixed values for the lower 96 bits:

```
u=bt.uuid(12345);
print u
→ [12345,4096,2147483776,1604007163]
for v in u do print hexstr(v) end
→ 3039
  1000
  80000080
  5f9b34fb
```

A few of the standard 32 bit UUIDs are:

Hex	Decimal	Service Class
3	3	RFCOMM
100	256	L2CAP
1101	4353	Serial Port
1103	4355	Dialup Networking
1105	4357	Obex (Object Exchange)
1111	4369	Fax
1204	4612	Generic Telephony

- **RFCOMM (Radio Frequency Communications):** Provides reliable communication between two Bluetooth devices. This corresponds to the TCP layer in the Internet world.
- **Channel:** An integer identifying an RFCOMM communication stream. This corresponds to a port number in the Internet world. A service can be reached by a device address and a channel number.

Connections Are Streams

Once created, a Bluetooth connection is accessed via module io (p. 36):

- `io.read`, `io.readln`, and `io.readm` receive data,
- `io.write`, `io.writeln`, `io.writem`, `io.print`, and `io.println` send data,
- `io.avail` gets the number of bytes which can be read without blocking,
- `io.wait` waits for data which can be read without blocking,
- `io.close` closes the connection.
- `io.ces` gets and sets the character encoding scheme. As with files, the default is `io.raw`.
- `io.timeout` sets the timeout for send and receive operations.
- `io.flush` sets the auto flush state. If auto flushing is disabled, `io.flush` must be called to make sure all data is sent.

Simple Example

To illustrate use of the **m** Bluetooth module, a trivial client-server example is presented. The server reverses each line of input it receives.

Client code:

```
use bt, io
// have the user select a device
dev=bt.select();
// connect to server
s=bt.conn(dev["adr"], "Reverser");
// write a line
io.writeln(s, "Hello world!");
// read the result
print io.readln(s)
→ !dlrow olleH
// and again
io.writeln(s, "Bye server");
print io.readln(s)
→ revres eyB
io.close(s)
```

Server code:

```
// a function which reverses a string
function reverse(s)
  c=code(s);
  i=0; j=len(c)-1;
  while i<j do
    h=c[i]; c[i]=c[j]; c[j]=h; i++; j--
  end;
  return char(c)
end

use bt, io
// create and advertise a service called "Reverser"
service=bt.start("Reverser");
while true do // loop forever
  // wait for a client
  io.print(io.stdout, "Waiting...");
  s=bt.accept(service);
  print bt.adr(s),"ok.";
  // read each line, writing it back reversed
  line=io.readln(s);
  while line#null do
    io.writeln(s, reverse(line));
    line=io.readln(s)
  end;
  io.close(s)
end
→ Waiting...00:0E:07:C9:EE:88 ok.
   Waiting...
```

bt.accept

- `function accept(service)→ Native Object`

  Permissions: `FreeComm`

Marks `service` available, then waits for a device connecting to `service`. When a device connects successfully, marks `service` as unavailable, and returns the connection stream.

See `bt.start` (p. 133) for an example.

bt.adr

- `function adr(stream)→ String`

  Permissions: `FreeComm`

- `function adr()→ String`

  Permissions: `FreeComm`

With one argument, returns the Bluetooth address of the device `stream` is connected to.

Without arguments, returns the local (own) Bluetooth address.

```
s=bt.accept(service);
// who connected?
print bt.adr(s)
→ 00:0E:07:C9:EE:88
// our own bluetooth address
print bt.adr()
→ 00:E0:03:5E:AF:CD
```

bt.chan

- `function chan(service)→ Array`

  Permissions: `FreeComm`

- `function chan(adr, uuid)→ Array`

  Permissions: `FreeComm`

With one argument, returns the channel number of `service`, in an array with the service name as key.

With two arguments, queries the service discovery database of the device with address `adr` for all services with the service class UUID defined by `uuid`, and returns their channel numbers in an array with the service names as keys. See `bt.uuid` (p. 135) for the values allowed for `uuid`.

```
// create a service on a fixed channel
s=bt.start("Sample", 18);
// obtain the channel of the service
c=bt.chan(s);
print c, keys(c)
→ [18] [Sample]
// query a device for all Obex services
c=bt.chan("00:0E:07:C9:EE:88", 4357);
print c, keys(c)
→ [9] [OBEX Object Push]
// query a device for all services using RFCOMM
c=bt.chan("00:0E:07:C9:EE:88", 3);
print c, keys(c)
→ [1,2,10,9,15,11,12,3] [Hands-Free Audio Gateway,
   Headset Audio Gateway,OBEX File Transfer,OBEX Object
   Push, Imaging,SyncMLClient,...<8>]
```

bt.conn

- `function conn(adr, uuidOrChannel)→ Native Object`

  Permissions: `FreeComm`

If `uuidOrChannel` is an array or a string, queries the service discovery database of the device with address `adr` for the first service with the service class UUID defined by `uuidOrChannel`, then connects to the service's channel.

If `uuidOrChannel` is a number, connects directly to channel `uuidOrChannel` of the device with address `adr`, without querying the database.

```
// connect to the Obex service on a device
dev="00:0E:07:C9:EE:88";
s=bt.conn(dev, [4357]);
io.close(s)
// connect to channel 18 on the same device
s=bt.conn(dev, 18);
io.close(s)
```

bt.name

- `function name()→ String`

  Permissions: `FreeComm`

- `function name(newname)→ String`

  Permissions: `FreeComm+WriteApp`

Without an argument, returns the local (own) device name. With a single argument, set the local device name to `newname` and returns the old name.

```
// change the name, returning the old one
print bt.name("Test Device #1")
→ Nokia 6670
// get the current name
print bt.name()
→ Test Device #1
```

bt.scan

- `function scan(limited=false)→ Array`

  Permissions: `FreeComm`

- `function scan()→ Array`

  Permissions: `FreeComm`

With a single argument, scans for other visible bluetooth devices in the neighborhood, and returns the first device found, or `null` if there is no visible device.
If `limited=false`, the scan is performed with general unlimited inquiry access code (IAC), returning all devices.
If `limited=true`, the scan is performed with the faster limited IAC, but only returning devices which are scanning with limited IAC.

Without an argument, continues scanning, and returns the next device found, or `null` if there are no more devices.

Making an SDP request (`bt.chan`, `bt.conn`) ends the current scan, i.e. the next call to `bt.scan` will always start a new scan.

Each device found is returned as an array with the following keys:

Key	Meaning	Type
adr	Device address	String
name	Device name	String
class	Device class	Integer

```
dev=bt.scan(false);
// print each device
while dev#null do
  print dev;
  // get the next device
  dev=bt.scan()
end
→ [00:E0:03:5E:AF:CD,Test Device #1,5243404]
→ [00:0E:07:C9:EE:88,Test Device #2,5251596]
```

bt.select

- function select()→ Array

  Permissions: FreeComm

Shows an interactive dialog scanning for Bluetooth devices and allowing the user to select one. Returns the selected device in the same format as returned by bt.scan (p. 132), or null if the user cancelled the selection.

```
print bt.select()
→ [00:E0:03:5E:AF:CD,Test Device #1,5243404]
```

bt.start

- function start(name, uuidOrChannel=null,flags=0)→
  Native Object

  Permissions: FreeComm

Creates a service with name name and returns it. To accept an incoming connection on the service, use bt.accept (p. 129).

If uuidOrChannel is an array or a string, bt.start finds an unused channel and creates a service with the UUID defined by uuidOrChannel. The service is advertised in the service discovery database of the device.

uuidOrChannel=null is equivalent to uuidOrChannel=name.

If `uuidOrChannel` is a number, listens directly on channel `uuidOrChannel`, without advertising the service.

The security imposed on incoming connections is defined by `flags`, which is a combination of the following values:

- `const` **`authenticate`** `= 1` Connecting devices must be paired, or mutual password authentication is requested.
- `const` **`encrypt`** `= 2` Data transfers are encrypted.
- `const` **`authorise`** `= 4` The user is asked for authorisation whenever a device attempts to connect to the channel.

```
// create a service with the UUID of the Fax
// service class, and asking for authorisation
service1=bt.start("My Fax", [4369], bt.authorise);
// wait for a connection
conn=bt.accept(service1);
...
// create a service listening on channel 18
service2=bt.start("Sample", 18);
conn2=bt.accept(service2);
...
```

bt.stop

- `function stop(service)→ null`

  Permissions: `FreeComm`

Stops `service`. If it has been advertised, it is removed from the service discovery database.

bt.timeout

- `function timeout()→ Number`

  Permissions: `FreeComm`

- `function timeout(ms)→ Number`

  Permissions: `FreeComm`

Gets or sets the timeout used during most functions of this module. Without arguments, 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. Bluetooth operations can block indefinitely, or use a timeout defined by the underlying system.

Throws ExcValueOutOfRange if ms exceeds 2147483 (35 minutes and 47.483 seconds).

The timeout is used in all following calls: whenever an operation does not complete within the given number of milliseconds, it throws ErrTimedOut.

```
 // allow 10 seconds to connect
bt.timeout(10000);
try
  s=bt.conn("00:E0:03:5E:AF:CD", 4)
  // connection successful...
catch e by
  if index(e, "ErrTimedOut") # 0 then throw e end;
  print "Could not connect within 10 seconds"
end
```

bt.uuid

- function uuid(uuid)→ Array

  Permissions: FreeComm

Converts a number, string or array to a 128 bit UUID, and returns the UUID as an array of four integers.

- If uuid is a number, uuid is considered a 32 bit Bluetooth UUID.
- If uuid is an array with one element, its only element is considered a 32 bit Bluetooth UUID.
- If uuid is an array with four elements, they are considered the four 32 bit values making up the entire 128 bit UUID (from highest to lowest).
- If uuid is a string with two characters or less, the characters are considered a 16 bit Bluetooth UIID.

- If `uuid` is a string with three or four characters, the characters are considered a 32 bit Bluetooth UIID.
- If `uuid` is a string with more than four characters, its first 16 characters are considered the 16 bytes of the UUID (from highest to lowest). Missing bytes are assumed zero.

All other values throw `ErrArgument`.

```
print bt.uuid(12345);
→ [12345,4096,2147483776,1604007163]
print bt.uuid([12345]);
→ [12345,4096,2147483776,1604007163]
print bt.uuid("Sample")
→ [1398893936,1818558464,0,0]
print bt.uuid([1,2])
→ ErrArgument thrown
```

bt.visible

- `function visible()→ Boolean`

  Permissions: `FreeComm`
- `function visible(newvisible)→ Boolean`

  Permissions: `FreeComm+WriteApp`

  Capabilities: `certified`

Compatibility of function `bt.visible`	
Nokia phones before Symbian 8[a]	ok
Nokia phones with Symbian 8[b]	`ErrNotSupported`
Symbian 3rd Edition and Sony Ericsson phones[c]	`ErrNotSupported`

[a] Changing the visibility is not reflected in the phone's settings UI.
[b] Parts of the Bluetooth API are not available on these phones.
[c] The visibility flag can only be read, but not set.

Without an argument, returns the current visibility state of this device: `true` if the device is detectable by others, `false` if it is not visible.

With an argument, sets the visibility to `newvisible`, and returns the old visibility state.

```
// make the device visible
bt.visible(true)
// is it visible?
print bt.visible()
→ true
```

6.2 Module comm: Serial Communications

This module provides access to the (usually software emulated) serial ports of the phone. Each communications device capable of performing serial communications is identified by its name (''module name'' in Symbian OS). Serial communication is often emulated by a device capable of multiplexing, a device may offer multiple units.

Often available devices are:

Device	Name	Unit
USB Serial Port	`ecacm`	`1`
Infrared (IrDA)	`ircomm`	`0`
Bluetooth Serial Port	`btcomm`	`0`

Note that not all devices are available on all phones, e.g. because the hardware or the appropriate driver are missing. Furthermore, the names and available units may also differ between phone models. Some units may already be used by the system software on the phone. A bit of try and error may be required to create a working connection.

Because of all these restrictions, use of serial communications should be avoided in **m** applications designed to be portable.

Serial Ports Are Streams

Once created, a serial port is accessed via module `io` (p. 36):

- `io.read`, `io.readln`, and `io.readm` receive data,
- `io.write`, `io.writeln`, `io.writem`, `io.print`, and `io.println` send data,

- `io.avail` gets the number of bytes which can be read without blocking,
- `io.wait` waits for data which can be read without blocking,
- `io.close` closes the connection.
- `io.ces` gets and sets the character encoding scheme. As with files, the default is `io.raw`.
- `io.timeout` sets the timeout for send and receive operations.
- `io.flush` sets the auto flush state. If auto flushing is disabled, `io.flush` must be called to make sure all data is sent.

comm.config

- `function config(port)→ Array`
- `function config(port,config)→ Array`

Permissions: `FreeComm`

With one parameter, gets the configuration of the serial port `port.port` must have been obtained via `comm.open` (p. 139).

With two parameters, sets the configuration from the fields in the array `config`, and returns the old configuration.

Configurations are represented by an array with the following elements:

Key	Meaning	Type
`bps`	Speed of the port (bits per second)	Integer
`data`	Number of data bits (5 to 8)	Integer
`stop`	Number of stop bits (1 to 2)	Integer
`parity`	Parity bit (0 to 4 corresponding to none, even, odd, mark, space)	Integer
`terms`	Characters considered terminators	String

```
// open an infrared port
s=comm.open("ircomm", 0)
// set it to 4 Mbit/s, 8 data and 1 stop bit, no parity
print comm.config(s, ["bps":4000000, "data":8,
                      "stop":1, "parity":0])
→ [9600,8,1,0,]
```

comm.link

- `function link(port,timeout=-1)→ Boolean`

Attempts to establish a link open for reading and writing, waiting until the other end allows writing. Returns `true` as soon as the link is up.

If `timeout>=0` and `timeout` milliseconds have passed without the link coming up, `false` is returned.

Throws `ExcValueOutOfRange` if `timeout` exceeds 2147483 (35 minutes and 47.483 seconds).

Reading from or writing to the port will also establish a link.

See `comm.signal` (p. 140) for an example.

comm.open

- `function open(name,unit,dceRole=false)→ Native Object`

  Permissions: `FreeComm`

Opens a serial port for communication over the device with name `name`, using unit `unit`, and returns a stream object representing the port. `unit` must be in the range returned by `comm.units` (p. 141). If `dceRole=false`, the serial port is working as a data terminal equipment; if `dceRole=true`, it is working as a data computer equipment.

Throws `ErrNotFound` if a device with this name does not exist. Throws `ExcIndexOutOfRange` if the unit is not within the range returned by `comm.units` (p. 141).

The following example communicates with a PC over the USB cable. If you use a terminal application on the PC communicating over the corresponding port[1], the tiny program will echo every line typed into the terminal application, converted to uppercase.

[1] On Windows, the port number (e.g. `COM3`) can usually be found in the hardware manager.

```
// Open a port to communicate with a PC
s=comm.open("ecacm", 1);
while true do
  l=io.readln(s);
  io.writeln(s, upper(l))
end
```

comm.signal

- `function signal(port)→ Number`
- `function signal(port,signals,mask=0x3f)→ Number`

Permissions: `FreeComm`

With one parameter, gets the (input) signals of the serial port `port`. `port` must have been obtained via `comm.open` (p. 139).

With two parameters, sets the (output) signals of the serial port contained in `mask` to the corresponding bit values in `signals`.

The signal bits are:

- `const` **`cts`** `= 1` Clear to send signal (input).
- `const` **`dcd`** `= 4` Data carrier detect signal (input).
- `const` **`dsr`** `= 2` Data set ready signal (input).
- `const` **`dtr`** `= 32` Data terminal ready signal (output).
- `const` **`rts`** `= 16` Ready to send signal (output).

```
// open an infrared port
s=comm.open("ircomm", 0)
// wait until a connection appears
print comm.link(s)
→ true
// wait until the connection disappears again
while comm.signal(s) & comm.dsr # 0 do
  sleep(1000); print comm.signal()
end
→ 3
  ...
  3
  0
```

comm.units

- `function units(name)→ Array`

  Permissions: `FreeComm`

Gets the range of units the device with name `name` supports. The range is returned as `[minUnit,maxUnit]`.

Throws `ErrNotFound` if a device with this name does not exist.

```
// Get the units of the IrDA device
print comm.units("ircomm")
→ [0,15]
```

6.3 Module net: TCP/IP Networking

This module supports creation of active TCP connections to hosts anywhere on the Internet. Secure connections based on SSL or TLS are also supported, as well as simple host name and IP address resolution.

Listening for incoming (passive) connections is also possible. Keep in mind that this generally only makes sense for local connections to `127.0.0.1`, as the phone is usually part of a private network and not visible to the rest of the internet.

This module does not support IPv6.

Connections Are Streams

Once created, a TCP/IP connection, whether secure or unsecure, is accessed via module `io` (p. 36):

- `io.read`, `io.readln`, and `io.readm` receive data,
- `io.write`, `io.writeln`, `io.writem`, `io.print`, and `io.println` send data,
- `io.avail` gets the number of bytes which can be read without blocking,
- `io.wait` waits for data which can be read without blocking, or for an incoming connection,

- `io.close` closes the connection or listening socket.
- `io.ces` gets and sets the character encoding scheme. As with files, the default is `io.raw`.
- `io.timeout` sets the timeout for send, receive and wait operations.
- `io.flush` sets the auto flush state. If auto flushing is disabled, `io.flush` must be called to make sure all data is sent.

Internet Access Points

Except for local connections, using TCP/IP requires the phone to connect to an IAP (Internet Access Point), typically via GPRS or UMTS, or via WLAN on devices supporting it. The TCP/IP functions of the phone deal with these automatically, depending on the phone configuration. The `net` module provides limited support to manage IAP connections: see `net.iap` (p. 145), `net.iaps` (p. 146), `net.start` (p. 151) and `net.stop` (p. 151).

net.accept

- `function accept(pstream)→ Native Object`

  Permissions: `CostComm`

Waits for a new incoming connection on the port defined by `pstream`, and returns it as a stream.

`pstream` is a passive stream which is obtained by call to `net.listen` (p. 147). See there for a complete example.

net.adr

- `function adr(hostname)→ Array`

  Permissions: `CostComm`

Resolves a host name to its IP address or addresses. The addresses are returned as an array of strings, each string representing the IP address in the standard dot notation.

```
print net.adr('www.google.com')
→ [216.239.59.103,216.239.59.104,
   216.239.59.99,216.239.59.147]
```

See also: net.local (p. 149), net.remote (p. 150)

net.cert

- function cert(stream)→ Array

  Permissions: CostComm

Gets the X.509 server certificate of the secure connection stream. The certificate identifies and (if it is valid) authenticates the host the connection has been made to.

This function returns null if stream is not secure.

The certificate is returned as an array with the following keys:

Key	Meaning	Type
subject	Certified subject (in X.500 format)	Array
issuer	Certificate issuer (in X.500 format)	Array
version	Certificate version	Integer
serial	Certificate serial number	String
start	Start of validity period	Seconds
end	End of validity period	Seconds
md5	Fingerprint of certificate (MD5 hash)	String

subject and issuer are arrays containing key-value pairs, with the keys being hierarchical OID numbers. For instance, the key "2.5.4.3" stands for Common Name, and "2.5.4.10" for Organization Name.

start and end define the validity period of the certificate, in seconds since year zero, as used by module time (p. 50).

serial and md5 encode each byte as a string character; use .code (p. 8) to convert them to single bytes.

```
// connect to a secure Web server
s=net.conn("www.yellownet.ch", 443, net.ssl);
// send a request
io.write(s, 'GET / HTTP 1.1\r\n\r\n');
// read the first four lines
for i=1 to 4 do
  print io.readln(s)
end
→ HTTP/1.1 302 Found
   Date: Tue, 24 May 2005 12:47:08 GMT
   Server: Stronghold
   Location: https://www.postfinance.ch/
// look at the certificate
c=net.cert(s);
print c["subject"]["2.5.4.3"]
→ www.yellownet.ch
print c["subject"]["2.5.4.10"]
→ Die Schweizerische Post
// close the connection
io.close(s)
```

net.conn

- `function conn(host, port, secure=null, silent=false, authName=null)→ Native Object`

Permissions: `CostComm`

Connects to the host `host` on TCP/IP port `port`. `host` can be a host name (e.g. `"www.m-shell.net"`), or an IP address (e.g. `"212.117.205.10"`).

If `secure=null`, the connection is unsecure. To secure the connection, use one of the following constants:

- `const` **`ssl`** `= "SSL3.0"` Use SSL (Secure Sockets Layer) 3.0.
- `const` **`tls`** `= "TLS1.0"` Use TLS (Transport Layer Security) 1.0.

If `silent=false`, the user will be prompted when the certificate presented by the server cannot be authenticated or has expired, giving the user the opportunity to accept the certificate for this session.

If `silent=true`, an invalid certificate will simply throw `ErrCertificateUnknown`, or some other SSL exception.

`authName` indicates the expected name authenticated by the certificate.

If authName=null, it defaults to host.

Compatibility of Secure Connections	
Sony Ericsson phones	Unreliable, may hang

```
// connect to airbit's SMTP mail server
s=net.conn("mail.airbit.ch", 25);
// read the prompt
print io.readln(s)
→ 220 mail.airbit.ch ESMTP ...
// immediately logout again
io.write(s, "QUIT\r\n");
// read the goodbye message
print io.readln(s)
→ 221 Service closing transmission channel
// close the connection
io.close(s)
```

For a secure connection example, see net.cert (p. 143).

net.iap

- function iap()→ Array

  Permissions: CostComm

- function iap(setting)→ Array

  Permissions: CostComm+WriteApp

  Capabilities: extended

Sets and gets the preferred Internet Access Point (IAP) to use. The preferred IAP setting consists of an array with three elements:

Index	Meaning	Type
0	Prompt user for IAP when connecting	Boolean
1	Preferred IAP id	Number
2	Bearer types supported by this setting	Number

The preferred IAP id corresponds to an id of the IAP table in the phone, as returned by net.iaps. The bearer set defines the set of bearer types supported by this setting.

Without arguments, this function returns the current preferred IAP setting. With a single boolean argument, it returns the old setting and sets the ''prompt user'' flag. With an array argument, it updates the corresponding entries, depending on the length of the array (1 to 3 elements).

```
// get current setting
s=net.iap();
print s
→ [false,14,3]
// change the preferred IAP id to 2 and enable prompting
net.iap([true, 2])
// disable prompting
net.iap(false)
// restore the old setting
net.iap(s)
```

net.iaps

- `function iaps(bearerMask=net.csd|net.wcdma|net.lan|net.cdma2000|net.virtual)→ Array`

Permissions: `CostComm`

Returns the configured IAPs whose bearer matches one of the flags in `bearerMask`. The mask bits are the following:

- `const` **`csd`** `= 1` circuit switched data: slow dialup connection.
- `const` **`wcdma`** `= 2` wide band code division multiple access, a 3G (UMTS) technology; also includes GPRS if 3G is not available.
- `const` **`lan`** `= 4` local area network, typically WLAN.
- `const` **`cdma2000`** `= 8` code division multiple access 2000, another 3G technology.
- `const` **`virtual`** `= 16` virtual bearer using another transport.

Each array element returned is itself an array with the following fields:

Key	Meaning	Type
`id`	Internal IAP id	Number
`name`	IAP name	String

```
// get all IAPs
for p in net.iaps() do
  print p
end
→ [1,Easy WLAN]
   [2,Swisscom GPRS]
   [3,Swisscom MMS]
   [4,Swisscom Internet]
   [5,Airbit WLAN]
// get only local network (WLAN) IAPs
for p in net.iaps(net.lan) do
  print p
end
→ [1,Easy WLAN]
   [5,Airbit WLAN]
// use the last WLAN for further connections
net.iap([false,p['id']])
```

net.listen

- `function listen(port, addr='0.0.0.0', queue=4)→ Native Object`

Creates a *passive stream* listening for incoming connections on the given port and address, and returns it. The address `0.0.0.0` allows connections to any valid IP address of the device. `queue` is the maximum number of queued unaccepted connections.

The passive stream can be passed to the following functions:

- `net.accept` (p. 142) waits for an incoming connection, and returns it.
- `io.wait` (p. 44) allows to simultaneously wait for an incoming connection and data being available on established connections or streams. If `io.wait` returns a passive stream, `io.avail` (p. 38) on this stream will return `1` afterwards.
- `io.close` (p. 39) closes the passive stream and stops listening on the given port, freeing it for other processes.

The following example is a server waiting for connections on port 4242,

receiving lines from the incoming connections and sending them back reversed. You may want to compare this example to the corresponding Bluetooth implementation in section 6.1 (p. 128). The main difference is that TCP/IP supports multiple connections per port and thus requires `io.wait` to manage them simultaneously.

```
use net, io, array
// create a passive stream listening on port 4242
p=net.listen(4242);
m=[p]; // the monitored streams
while true do // loop forever
  // wait for a client
  io.print(io.stdout, "Waiting...");
  s=io.wait(m);
  if s=p then // new connection, accept it
    print "got new connection.";
    append(m, net.accept(p))
  else // data on existing connection
    io.print(io.stdout, "reading...");
    line=io.readln(s);
    if line#null then
      print "got", line;
      io.writeln(s, reverse(line))
    else // EOF, remove connection
      print "lost connection.";
      io.close(s);
      array.remove(m, array.index(m, s))
    end
  end
end
→ Waiting...got new connection.
   Waiting...got new connection.
   Waiting...reading...got Lucky Luke
   Waiting...reading...got Jolly Jumper
   Waiting...reading...lost connection.
   Waiting...
```

Sample client calls producing the above output are:

```
s1=net.conn('127.0.0.1', 4242);
s2=net.conn('127.0.0.1', 4242);
io.write(s1, 'Lucky Luke\n');
io.readln(s1)
→ ekuL ykcuL
io.write(s2, 'Jolly Jumper\n');
io.readln(s2)
→ repmuJ ylloJ
io.close(s1)
```

net.local

- `function local(stream)→ Array`

  Permissions: `CostComm`

Returns the local (your own) IP address and port of the connection defined by `stream` as an array:

Key	Meaning	Type
`adr`	TCP/IP address	String
`port`	TCP/IP port	Number

Throws `ErrBadHandle` if `stream` is not an open connection.

```
s=net.conn('www.post.ch', 80);
print net.local(s)
→ [10.122.18.7,32803]
```

net.name

- `function name(address)→ Array`

  Permissions: `CostComm`

- `function name()→ Array`

  Permissions: `CostComm`

Finds the host names belonging to an IP address. The IP address must be a string in standard dot notation. The names are returned as an array of strings.

Without arguments, returns the local (own) host name.

```
print net.name('62.65.129.6')
→ [mail.infowing.ch]
print net.name()
→ [localhost]
```

net.remote

- `function remote(stream)→ Array`

  Permissions: `CostComm`

Returns the remote (the remote host's) IP address and port of the connection defined by `stream` as an array:

Key	Meaning	Type
`adr`	TCP/IP address	String
`port`	TCP/IP port	Number

Throws `ErrBadHandle` if `stream` is not an open connection.

```
p=net.listen(4242);
s=net.accept(p);
print net.remote(s)
→ [127.0.0.1,39934]
```

net.shut

- `function shut(stream, abort=false)→ null`

  Permissions: `CostComm`

Shuts the connection defined by `stream` down. If `abort=false`, shutdown is gracefully, i.e. all pending data is transmitted. If `abort=true`, sending and receiving is stopped immediately.

`io.close` (p. 39) also shuts down a connection, but `net.shut` gives finer control over connection termination, and allows to catch errors.

```
s=net.conn('mail.airbit.ch', 25);
// abort the connection
net.shut(s, true)
```

net.start

- `function start()→ null`

  Permissions: `CostComm`

- `function start(prompt)→ null`

  Permissions: `CostComm`

Starts the IAP connection.

Without argument, connects using the current IAP settings. This is normally not required, as connections are created on demand.

With argument, connects using the current IAP settings, but overrides the `prompt` flag: if `prompt=false`, the user is not prompted to choose an IAP; if `prompt=true`, the user is always prompted.

```
// start the connection without prompting for an IAP
net.start(false)
```

net.stop

- `function stop()→ null`

  Permissions: `CostComm`

  Capabilities: `certified`

Stops the current IAP connection. Calling this function is normally not required, as connections are removed when they are no longer needed.

```
// change the IAP, then stop the connection
net.iap([false, 7]);
net.stop()
// resolving a host name should restart the connection
// with the new IAP
net.adr('www.m-shell.net')
→ [62.202.44.142]
```

net.timeout

- `function timeout()→ Number`

  Permissions: `CostComm`

- `function timeout(ms)→ Number`

  Permissions: `CostComm`

Gets or sets the timeout used when looking up names and when connecting. Without arguments, 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. TCP/IP operations can block indefinitely, or use a timeout defined by the underlying system.

Throws `ExcValueOutOfRange` if `ms` exceeds 2147483 (35 minutes and 47.483 seconds).

The timeout is used in all following name resolution, connect and shutdown calls: whenever an operation does not complete within the given number of milliseconds, it throws `ErrTimedOut`.

```
 // give the phone 10 seconds to connect
net.timeout(10000);
try
  s=net.conn("mail.airbit.ch", 25)
  // connection successful...
catch e by
  if index(e, "ErrTimedOut") # 0 then throw e end;
  print "Could not connect within 10 seconds"
end
```

7. Messaging

7.1 Module mms: Multimedia Messages

Compatibility of module mms	
Sony Ericsson phones: all functions except mms.send[a]	ErrNotSupported

[a]The MMS API on SE devices only supports sending. Use module msg to read MMS.

This module supports sending and receiving of multi media messages (MMS). In the context of this module, an MMS is simply a set of files being sent from and to mobile devices, very similar to an e-mail with attachments.

MMS 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 mms.receive will return these numbers. Messages received earlier can be retrieved from the inbox.

The typical sequence to consume messages starting with a certain token in the subject (//tok in our example) is:

```
nr=mms.receive(); // wait for a new message
msg=mms.get(nr); // get the message
words=split(msg["subject"]); // split into words
if len(words)>0 and words[0] = "//tok" then
  // first word is //tok, process message files
  for f in msg["files"] do
    ...
  end;
  // delete it from the inbox
  mms.delete(nr)
end
```

The functions in this module correspond to those in module sms (p. 167) for short messages.

mms.delete

- function delete(msgnum)→ null

  Permissions: FreeComm+WriteApp

Delete the message with number msgnum from the inbox.

Throws ErrNotFound if the message with this number does not exist.

```
// delete all MMS inbox messages older than a week
lastweek=time.get()-7*24*3600;
for id in mms.inbox() do
  if mms.get(id)["time"]<lastweek then
    mms.delete(id)
  end
end
```

mms.get

- function get(msgnum)→ Array

  Permissions: FreeComm+ReadApp

Compatibility of function mms.get
Symbian 3rd Edition phones: the file names returned by this call are not directly accessible, use mms.open (p. 155) to read their data.

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 (or other address) of the sender of the message.
`subject`	The subject of the message.
`time`	The time stamp of the message, as seconds since the start of year 0. See also module `time` (p. 50).
`unread`	`true` if the message is still unread, `false` if it has been seen.
`files`	The list of files comprising the message.

Throws `ErrNotFound` if the message with number `msgnum` does not exist.

```
// play all MIDI files found in the MMS inbox
for id in mms.inbox() do
  for f in mms.get(id)["files"] do
    if len(f)>3 and substr(f,len(f)-4)=".mid" then
      audio.play(f); audio.wait()
    end
  end
end
```

mms.inbox

- `function inbox()→ Array`

  Permissions: `FreeComm+ReadApp`

Gets the ids of all MMS messages in the inbox.

```
print mms.inbox()
→ [1045642,1045678,1047382]
```

mms.open

- `function open(msgnum,index)→ Native Object`

  Permissions: `FreeComm+ReadApp`

Opens the attachment with index `index`, and returns a stream object to read its data from. `index` is the index into `msg.get(msgnum)['files']`.

The returned stream can be accessed with most functions from module `io` (p. 36):

- `io.read`, `io.readln`, and `io.readm` read data,
- `io.size` gets the total number of bytes,
- `io.avail` gets the number of bytes remaining,
- `io.seek` changes the read position,
- `io.close` closes the stream,
- `io.ces` gets and sets the character encoding scheme.

```
// Copy all attachments of an MMS to a directory
function copyAttmts(msgnum, dir)
  m=mms.get(msgnum);
  for j=0 to len(m['files'])-1 do
    name=m['files'][j];
    name=substr(name, rindex(name, '\\')+1);
    i=mms.open(msgnum, j);
    print "Copying ",io.size(i),"bytes to",name;
    o=io.create(dir+'\\'+name);
    b=io.read(i, 256);
    while b#null do
      io.write(o, b); b=io.read(i, 256)
    end;
    io.close(i); io.close(o)
  end
end
```

mms.receive

- `function receive(timeout=-1)→ Number|null`

  Permissions: `FreeComm+ReadApp`

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`.

Throws `ExcValueOutOfRange` if `timeout` exceeds 2147483 (35 minutes and 47.483 seconds).

```
// quickly check whether there is a new MMS
id=mms.receive(0);
if id#null then
  msg=mms.get(id);
  // process msg
end
```

mms.send

- `function send(recipient, subject, files, sender=null)→ null`

  Permissions: `CostComm+Read(files)`

- `function send(recipients, subject, files, sender=null)→ null`

  Permissions: `CostComm+Read(files)`

Compatibility of function `mms.send`	
Sony Ericsson phones: character sets of attached files and the sender cannot be set.	`ErrNotSupported`

Sends a multimedia 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.

The message will get the subject `subject`. The files to be attached are defined by `files`, an array with one element for each file to be sent. Each element is:

- Either a string, directly denoting the file name, with automatically derived MIME type and default character set,
- or an array of one to three elements, in the form

`[name,mimeType,charset]`. `name` is a string denoting the file name, `mimeType` (if not missing or `null`) is the MIME type of the file, and `charset` (if not missing or `null`) is the character set/encoding specified as an integer IANA MIB enum value.

A few important character sets/encodings:

MIB enum	Description
3	US-ASCII
4	ISO-8859-1 (Latin 1)
5	ISO-8859-2 (Latin 2)
106	UTF-8
1000	ISO-10646-UCS-2 (''Unicode'')
1001	ISO-10646-UCS-4

If `sender` is not null, the `From:` field of the outgoing message is set to `sender`. Note that most MMSCs will set this field to the MSISDN of the sending device when receiving the MMS, so specifying a sender has no effect unless you operate your own MMSC.

This function throws `ErrNotFound` if any of the files to be attached does not exist.

This function returns as soon as the message has been placed in the outbox. Actual sending may occur at a later time (''store and forward'' principle).

```
// find all m scripts
f=files.scan(system.docdir + "*.m");
// prepend the directory
for i=0 to len(f)-1 do
  f[i]=system.docdir+f[i]
end;
// send all those files to two people
mms.send(["+41797654321", "+393401234567"],
         "My mShell scripts", f);
// send all those files again, specifying a MIME type
// and Latin 1 character set
for i=0 to len(f)-1 do
  f[i]=[f[i],'text/plain',4]
end;
mms.send(["+41797654321", "+393401234567"],
         "My mShell scripts", f);
```

mms.set

- function set(msgnum, message)→ null

  Permissions: FreeComm+WriteApp

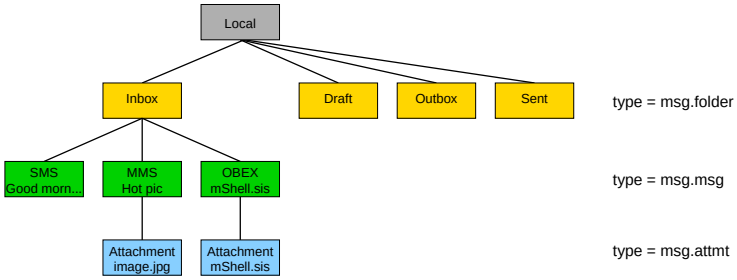
Updates the short message with number msgnum with the fields from message. The keys listed in mms.get (p. 154) must be used. The sender and subject of the message will only be changed in the MMS inbox summary; they cannot be changed in the actual message. files cannot be changed at all.

```
// mark all MMS in the inbox as unread
for id in mms.inbox() do
  mms.set(id, ["unread":true])
end
```

7.2 Module msg: Generic Message Access

This module provides generic access to the messages (e.g. SMS, MMS, OBEX) stored on the phone.

The phone organizes messages into a hierachical tree of *entries*, much like an ordinary file system. The most important entry types are folders, messages and attachments. A typical message hierarchy could look as follows:



Each entry is identified by its unique id[1]. There are module constants for the ids of the four main folders: msg.inbox, msg.draft, msg.outbox,

[1] These ids are the same as the ones used in module mms and module sms

and msg.sent.

Note that the organization of folders is device dependent. For instance, Sony Ericsson phones devices have dedicated service entries for MMS. Scan from msg.root to obtain the complete hierarchy.

msg.delete

- function delete(entryOrId)→ null

  Permissions: FreeComm+ReadApp+WriteApp

Delete the message entry identified by entryOrId. entryOrId can be a complete entry, as returned by msg.scan, or simply an integer id.

Throws ErrNotFound if this entry does not exist.

```
for m in msg.scan(msg.sent, msg.msg) do
  msg.delete(m)
end
```

msg.move

- function move(entryOrId,newParentEntryOrId)→ null

  Permissions: FreeComm+ReadApp+WriteApp

Move the message entry identified by entryOrId from its current parent to the entry identified by newParentEntryOrId. The latter is typically a folder. Both parameters can be a complete entry, as returned by msg.scan, or simply an integer id.

Throws ErrNotFound if this entry does not exist.

```
// move all .SIS file messages from the inbox to draft
for m in msg.scan(msg.inbox, msg.msg, "*.sis") do
  msg.move(m, msg.draft)
end
```

msg.open

- function open(entryOrId,index=0)→ Native Object|null

  Permissions: FreeComm+ReadApp

Opens a message or attachment entry, and returns a stream object to read the data at the given index from it. For attachment entries, the data is the file data from the attachment with the given index. For other entries, it is the message body, which has always index 0.

Returns null if the entry has no data to read.

The returned stream can be accessed with most functions from module io (p. 36):

- io.read, io.readln, and io.readm read data,
- io.size gets the total number of bytes,
- io.avail gets the number of bytes remaining,
- io.seek changes the read position,
- io.close closes the stream,
- io.ces gets and sets the character encoding scheme. For messages, the default is io.utf16le and for attachments io.raw.

```
// Copy an attachment from the inbox to a file.
function copyAttmt(name,file)
  ms=msg.scan(msg.inbox, null, name);
  // if there is no message, throw an exception
  if len(ms)=0 then
    throw "No message "+name
  end;
  // get the first attachment of the message
  ms=msg.scan(ms[0], msg.attmt);
  // if there is no attachment, throw an exception
  if len(ms)=0 then
    throw "No attachment for "+name
  end;
  i=msg.open(ms[0]);
  print "Copying ",io.size(i),"bytes";
  o=io.create(file);
  b=io.read(i, 256);
  while b#null do
    io.write(o, b); b=io.read(i, 256)
  end;
  io.close(i); io.close(o)
end
```

msg.scan

- `function scan(parent=msg.inbox,type=null,pattern="*")→ Array`

  Permissions: `FreeComm+ReadApp`

Scan the message entry identified by `parent` for its direct children. `parent` can be a complete entry, as returned by this function, or simply an integer id. `type` restricts the entry type to the given type (e.g. `msg.msg`). If `type=null`, entries of all types are returned. `pattern` is a pattern which the two entry descriptions must match. It is not case sensitive and can contain the wildcards `*` and `?`.

Returns an array with one element for each member found, each element being an array with the following keys:

Key	Meaning	Type
id	Entry id	Integer
descr	Entry description (E.g. start of message text)	String
descr2	Other description (E.g. message sender)	String
time	The time stamp of the message, as seconds since the start of year 0. See also module time (p. 50).	Number
unread	true if the message is still unread, false if it has been seen.	Boolean
type	Entry type	Integer

```
// count the messages in the inbox
print len(msg.scan()),"messages"
→ 13 messages
// get the sent messages containing "morning"
for m in msg.scan(msg.sent, msg.msg, "*morning*") do
  print m
end
→ [1048747,Good morning!,0779696969,63348191631,false,
   268439402]
   [1048748,This morning I can't see you,0797654321,
   63348191796,false,268439402]
```

msg Constants

- const **attmt** = The type indicating an attachment entry.
- const **draft** = The id of the folder with draft messages.
- const **folder** = The type indicating a folder entry.
- const **inbox** = The id of the folder with incoming messages.
- const **local** = The id of the service with local folders.
- const **msg** = The type indicating a message entry.
- const **outbox** = The id of the folder with outgoing messages.
- const **root** = The id of the root of the message entry hierarchy.
- const **sent** = The id of the folder with sent messages.

7.3 Module obex: Object Exchange Client

This module supports sending and receiving of files via OBEX (Object Exchange) over a Bluetooth®link. The module provides the client side; most Bluetooth equipped devices have an OBEX server which can accept files (`put` operation of the client); some servers can also deliver files (`get` operation of the client).

See also module `bt` (p. 125).

Usage of this module typically follows this pattern:

```
function btsend(files)
  // have the user choose a device
  dev=bt.select();
  if dev#null then
    adr=dev['adr'];
    // connect after getting the channel for the
    // OBEX Push Service
    obex.conn(adr, bt.chan(adr, obex.uuid)[0]);
    // send all the files
    for f in files do
      obex.put(f)
    end;
    obex.close()
  end
end

// send three files
btsend(['sample.dat', 'moon.gif', 'bells.mp3'])
```

obex.close

- `function close()→ null`

  Permissions: `FreeComm`

Closes the connection to the server. Does nothing if there is no connection.

obex.conn

- function conn(adr, channel, password=null)→ String

  Permissions: FreeComm

Connects to the OBEX server on the host with Bluetooth address adr, on channel channel. If password#null, it will be used during OBEX authentication.

The channel is normally obtained by querying the hosts service discovery database via bt.chan (p. 130) for obex.uuid (p. 167).

If successful, returns the ''who'' name of the OBEX server.

```
dev="00:0E:07:C9:EE:88";
channel=bt.chan(dev, obex.uuid)[0];
print obex.conn(dev, channel)
→ peer2
```

obex.get

- function get(path, name=null)→ String

  Permissions: Write(path)+FreeComm

Gets (pulls) a file from the server, storing it in path. The object (or file) to be pulled is given by name. If name=null, it equals to path without any directory components.

Note that not all servers support file pulling.

Throws ErrDisconnected if the client is not connected.

```
// get a vCard into the cards directory
obex.get('\\cards\\William.vcf', 'OwnCard.vcf')
```

obex.path

- function path(name, create=false)→ null

  Permissions: FreeComm

Changes the directory on the server to name. If name="..", changes to the parent directory. If create=true, the directory is also created if it doesn't exist.

Note that not all servers support directories.

Throws `ErrDisconnected` if the client is not connected.

```
// change to directory 'images', creating it if required
path('images', true);
// change back to the parent
path('..')
```

obex.put

- `function put(path, name=null, type=null, description=null)→ null`

  Permissions: `Read(path)+FreeComm`

Puts (pushes) a file to the server, getting the data from `file`. The name of the file on the server is given by `name`, its MIME type by `type`. `description` is an optional description of the data for the server.

If `name=null`, it equals to `file` without any directory components.

If `type=null`, it is derived from the file extension for many important file types.

Throws `ErrDisconnected` if the client is not connected.

```
// send a screen shot to the server
obex.put("c:\\Nokia\\Images\\Fe_img\\Fescr(0).jpg",
         "myapp.jpg", "image/jpeg",
         "Screen shot of my app")
```

obex.timeout

- `function timeout()→ Number`

  Permissions: `FreeComm`

- `function timeout(ms)→ Number`

  Permissions: `FreeComm`

Gets or sets the timeout used during most functions of this module. Without arguments, 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. OBEX operations can block indefinitely, or use a timeout defined by the underlying system.

The timeout is used in all following calls: whenever an operation does not complete within the given number of milliseconds, it throws ErrTimedOut.

Throws ExcValueOutOfRange if ms exceeds 2147483 (35 minutes and 47.483 seconds).

A timed out call will always close the OBEX connection; obex.conn (p. 165) must be called to reconnect.

obex.who

- function who()→ String|null
  Permissions: FreeComm
- function who(name)→ String|null
  Permissions: FreeComm

Gets or sets the local ''who'' name for the next connection.

Without arguments, returns the current ''who'' name, or null if none is set. With one argument, returns the old name and sets the new name to name. Setting it to null disables sending the ''who'' name.

Some servers assume a special role if a certain name is presented. For most purposes, you do not need to set a ''who'' name.

obex.who must be called before obex.conn (p. 165).

```
// set the "who" name to 'peer1'
obex.who('peer1')
```

obex Constants

- const **uuid** = 4357 The standard BT UUID for the Obex Push Service.

7.4 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
```

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 `time` (p. 50).
`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`.

Throws `ExcValueOutOfRange` if `timeout` exceeds 2147483 (35 minutes and 47.483 seconds).

```
// 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. 169) 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
```

8. Multimedia

8.1 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, WAV format or AMR-NB format files.

To directly play an existing audio file, use `audio.play` (p. 178).

To play parts of a file or record to a file, use `audio.open` (p. 176), followed by calls to `audio.play` (p. 178), `audio.record` (p. 179) and `audio.stop` (p. 180).

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. 176) and `audio.pos` (p. 179) access them. `audio.cut` (p. 175) 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 on some devices. The behaviour when playing tones or sound during a phone call varies between devices; some will throw `ErrInUse`, others will simply mute the sound. UIQ devices generally do not support playing sounds when a phone call is in progress.

`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 `ExcValueOutOfRange` 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 audio.cut	
Sony Ericsson UIQ2 phones cannot truncate at the beginning, start=0 is mandatory.	ErrNotSupported
Sony Ericsson UIQ3 phones cannot truncate at all.	ErrNotSupported

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

Compatibility of function audio.dtmf	
Some Nokia phones do not properly produce DTMF tones, and may set the tone playing device into an erroneous state.	A call to audio.stop may be required to reset the device.

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)`

Compatibility of function `audio.open`	
Nokia phones and Sony Ericsson UIQ2 phones do not support AMR-NB format for recording.	`ErrNotSupported`
Sony Ericsson UIQ3 phones do not support WAV and AU formats for recording.	`ErrNotSupported`
Sony Ericsson UIQ3 phones cannot handle file suffixes other than `.amr` when recording.	`ErrNotFound`, `ErrNotSupported`

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.
- `const` **`amr`** `= 4` Create a file in AMR-NB format (Adaptive Multi-Rate, Narrow Band).

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 μ-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.

audio.amr only supports its own 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.

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 1.2 (p. 4)). 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. 181) 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 form 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 audio.record	
Sony Ericsson phones cannot record phone conversations	ErrInUse
Sony Ericsson phones do not reliably detect unsupported sample rates, resulting in mismatches between sampled and played rates.	
Sony Ericsson UIQ3 phones do not support any seeking when recording. Calls to audio.pos are meaningless when recording.	

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. 175) 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. 180) 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
```

8.2 Module cam: Onboard Camera

This module provides access to the onboard camera for still images. Pictures taken can be processed or saved by module module graph (p. 57).

Since the camera is a shared resource and consumes battery power, it must be turned on before use by cam.on (p. 185) and turned off afterwards by cam.off (p. 185). A typical example using the camera might look as follows:

```
// show the available image sizes
for s in cam.sizes() do
  print s
end
→ [1280,960]
  [640,480]
  [160,120]
// turn the camera on for 640x480 size images
cam.on(1)
// produce a dark, contrast rich picture
cam.bright(-20); cam.contrast(30)
→ 0
  0
// display a view finder close to the top left corner
cam.view(10,10)
// take an image
icon=cam.take()
// turn the camera off
cam.off()
// save the image via the graph module
s=graph.size(icon); // get the image size
graph.size(s[0], s[1]); // make graph big enough
graph.put(0,0,icon); // draw the image
graph.save("keyboard.jpg") // save it
```

Sample **m** *screen*

cam.bright

- function bright()→ Number
- function bright(b)→ Number

Gets or sets the brightness of the image taken. The brightness is a number between -100 (very dark) and 100 (very bright). Standard brightness is 0.

Without arguments, returns the currently used brightness. With one argument, returns the old brightness, and sets the new brightness to b.

Throws ErrInUse or ErrNotReady if the camera has not been turned on.

```
// show the view finder, increasing brightness
cam.on()
cam.view()
for b=-100 to 100 by 10 do
  cam.bright(b); sleep(1000)
end;
cam.off()
```

cam.contrast

- function contrast()→ Number
- function contrast(c)→ Number

Gets or sets the contrast of the image taken. The contrast is a number between -100 (minimum contrast) and 100 (maximum contrast). Standard contrast is 0.

Without arguments, returns the currently used contrast. With one argument, returns the old contrast, and sets the new contrast to c.

Throws ErrInUse or ErrNotReady if the camera has not been turned on.

```
// show the view finder, increasing contrast
cam.on()
cam.view()
for c=-100 to 100 by 10 do
  cam.contrast(c); sleep(1000)
end;
cam.off()
```

cam.index

- `function index()→ Number`
- `function index(camIndex)→ Number`

Compatibility of function `cam.index`	
Sony Ericsson UIQ3 phones (P990) have no public API for the front camera (index 1)	`cam.sizes` returns `[]`

On devices with more than one built-in camera, selects the camera to operate on. The camera index must be greater than or equal to 0 and less than `cam.count` (p. 188).

Without arguments, returns the index of the currently used camera. With one argument, turns the old camera off, returns the old index, and sets the new camera index.

By default, the first camera (index 0) is selected.

Throws `ExcIndexOutOfRange` if the camera does not exist.

```
// select the 2nd camera, if there is one
if cam.count> 1 then
  cam.index(1)
else
  print "No second camera"
end
```

cam.off

- `function off()→ null`

Removes the view finder if it is shown, and turns the camera off. Does nothing if the camera is already off.

cam.on

- `function on(sizeIndex=0)→ null`

Turns the camera on and prepares it for taking images of the size `cam.sizes()[sizeIndex]`.

Throws `ExcIndexOutOfRange` if `sizeIndex` is less than 0 or greater than the `cam.sizes() - 1`.

Throws `ErrInUse` if the camera is already on, or used by another application.

cam.sizes

- `function sizes()→ Array`

Returns the available image sizes, as an array of arrays containing image width and image height. The actual sizes returned are hardware dependent.

The camera does not have to be on to obtain the image sizes.

On some devices, the setting of the screen mode (`ui.mode` (p. 93)) has an effect on the available sizes. For instance, maximum resolution might only be available in landscape mode.

```
for s in cam.sizes() do
  print s
end
→ [640,480]
  [320,240]
  [160,120]
```

cam.take

- `function take()→ Native Object`
- `function take(jpegpath,quality=75)→ null`

  Permissions: `Write(jpegpath)`

Compatibility of function `cam.take`	
Sony Ericsson UIQ2 phones if `cam.view` has not been called.	`ErrInUse`

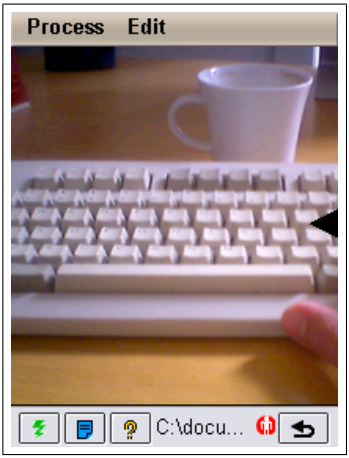
Without arguments, takes an image of the configured size, brightness and contrast and returns it as an icon (see `graph.icon` (p. 71)). The icon can be saved, scaled, or analyzed using functions in module `graph` (p. 57).

With one or two arguments, takes an image of the configured size, brightness and contrast and saves it directly to file `jpegpath`, compressing for the given `quality`. `quality` must be between 1 and 100. No icon is produced in this case.

Throws `ErrInUse` or `ErrNotReady` if the camera has not been turned on.

Throws `ExcValueOutOfRange` if the JPEG quality is out of range.

```
i=cam.take();
print i
→ icon@4186d8
// scale the image to one quarter and display it
graph.size(i,0.5)
→ [640,480]
graph.put(0,0,i)
graph.show()
```



Sample **m** *screen*

```
// take an image and save it to snapshot.jpg
cam.take('snapshot.jpg')
```

cam.view

- `function view(x=0,y=0,w=160,h=120)→ Array`

Compatibility of function `cam.view`
Sony Ericsson UIQ3 phones blacken the entire window when showing the view finder.

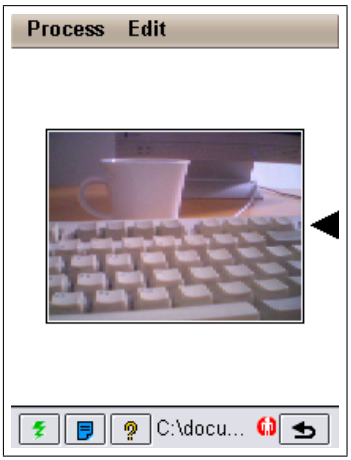
Shows a view finder (the image currently seen by the camera) on the screen at coordinates `(x,y)`, in a rectangle of roughly width `w` and height `h`. `(0,0)` is at the upper left corner of the **m** application view.

Returns the actual size of the rectangle used.

Throws `ErrInUse` or `ErrNotReady` if the camera has not been turned on.

Throws `ErrNotSupported` if the requested view finder size is not supported.

```
// show the view centered on the graph view
gs=graph.size();
cam.on();
vs=cam.view();
x=math.trunc((gs[0]-vs[0])/2);
y=math.trunc((gs[1]-vs[1])/2);
// draw a frame around the view
graph.rect(x-2,y-2,vs[0]+4,vs[1]+4);
graph.show();
cam.view(x, y)
```



Sample **m** *screen*

`cam` Constants

• `const` **`count`** `=` The number of cameras available. On some devices, some cameras may be unaccessible via this module.

8.3 Module `video`: Playing Videos

Compatibility of module `video`	
Sony Ericsson UIQ2 phones do not offer a public video playing API.	Source file for module not found

This module provides functions to play recorded videos on the device screen and audio. Supported video file formats depend on the device, but generally include standard MP4 formats.

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.

The video is shown on top of any other screen contents. Both the source region of the video frames and the destination region on the screen can be defined. Within certain limits, the video will then be scaled to match the requested regions.

A simple sequence to a play a video file is:

```
// open the video file
video.open('FasterThanMyShadow.mp4');
// start playing it
video.play();
// wait until playing has finished
video.wait()
```

video.busy

- `function busy()→ Boolean`

Returns `true` if the video is still playing. Returns `false` otherwise.

This function checks only the current **m** process: it will return `false` if another process is playing a video.

```
video.play();
while video.busy() do
  print "at",video.pos(),"ms";
  sleep(1000)
end;
print "video replay ended"
→ at 600 ms
   at 1600 ms
   at 2700 ms
   at 3800 ms
   video replay ended
```

video.close

- `function close()→ null`

Closes the currently accessed video file and frees all resources.

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

```
video.stop();
video.close()
```

video.hide

- `function hide()→ null`

Hides the video display. This can be called after `video.stop` to hide the last frame shown. If the video is playing, hiding results only in a short flicker or has no effect at all.

See also `video.show` (p. 192).

video.open

- `function open(file)→ Array`

  Permissions: `Read(file)`

Opens a file for playing, and returns an array describing the video:

Key	Meaning	Type
`len`	Duration of the video in milliseconds	String
`size`	Width an height of the video frames (pixels)	Array
`type`	MIME file type of the video	String
`fps`	Frame rate (frames per second)	Number
`audio`	Does the video have audio data	Boolean

The source region is set to the whole frame, and the destination region to the entire screen. See `video.view` (p. 193) for more information about source and destination regions.

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

```
v=video.open('FasterThanMyShadow.mp4');
for k in keys(v) do
  print k,v[k]
end
→ len 6919
   size [320,240]
   type video/MP4V-ES
   fps 24
   audio true
```

video.play

- function play()→ null

Starts or continues playing the currently open video file in the current view setting.

This function immediately returns, before playing completes. Exceptions can therefore be thrown anywhere in the following code. Use video.wait (p. 194) to wait for completion.

Throws ErrInUse if the video unit is busy playing another video.

Throws ErrNotReady if no file has been opened,

video.pos

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

Without arguments, returns the playing position in the current file, in milliseconds form 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
video.open('sample.mp4');
video.pos(5000);
video.play();
sleep(7000);
video.stop();
print video.pos()
→ 11600
```

video.show

- `function show()→ null`

Shows the last shown frame again. This can be useful if the video is stopped or paused and the screen has been changed otherwise, clearing or hiding the video display.

Throws `ErrNotReady` if no file has been opened.

See also `video.hide` (p. 190).

video.stop

- `function stop()→ null`

Stops (or pauses) the currently playing video at the current position, without hiding it.

video.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.

```
v=video.play("c:\\documents\\video\\HomeBox.mp4");
while video.busy() do
  sleep(100);
  if video.pos()>v["len"]-2000 then
    video.volume(video.volume()-5) // fade out
  end
end
```

video.view

- function view()→ Array
- function view(settings)→ Array

Compatibility of function video.view	
Rotating the video is not supported on all devices	rot is ignored

Gets or sets the source (video) and destination (screen) region settings for playing videos. Without an argument, returns the current settings. With one argument, sets the new settings according to the array elements found in settings, and returns the old settings.

Settings are specified as an array:

Key	Meaning	Type
src	Video rectangle to show ([x,y,w,h])	Array
dst	Screen rectangle to show video in ([x,y,w,h])	Array
rot	Rotate the video by this times 90 degrees	Number

```
v=video.open('FasterThanMyShadow.mp4');
// squeeze the video into the upper left quadrant
view=video.view();
view['dst'][2]=v['size'][0] / 2;
view['dst'][3]=v['size'][1] / 2;
video.view(view);
// show only the upper half of the video
r=[0,0,v['size'][0],v['size'][1]/2];
view.view(['src':r,'dst':r])
```

video.wait

- `function wait()→ null`

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

This function checks only the current **m** process: it will return immediately if a video is played by another process (inside or outside of **m**).

9. Telephony

9.1 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`

  Capabilities: `extended`

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[1]. 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 for GSM cells, and a number greater than 65535 on UTMS cells.

```
print gsm.cid()
→ 17437
```

[1] Usually, a single BTS (base transceiver station) covers multiple cells via sectorial antennas mounted on a single antenna tower.

gsm.net

- `function net()→ Array`

  Permissions: `ReadApp`

  Capabilities: `extended`

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

Key	Contents
`mcc`	Mobile Country Code (MCC)
`mnc`	Mobile Network Code (MNC)
`short`	Short Network Name
`long`	Long Network Name
`lac`	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. 197)).

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

```
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`

  Capabilities: `extended`

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.

Throws ExcValueOutOfRange if timeout exceeds 2147483 (35 minutes and 47.483 seconds).

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

Compatibility of function gsm.signal	
Sony Ericsson UIQ3 phones do not support this API.	Call returns 0.

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*

Capabilities: `extended`

Compatibility of constant `imsi`	
Nokia 6600: the IMSI cannot be obtained.	`imsi=000000000000000`

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
```

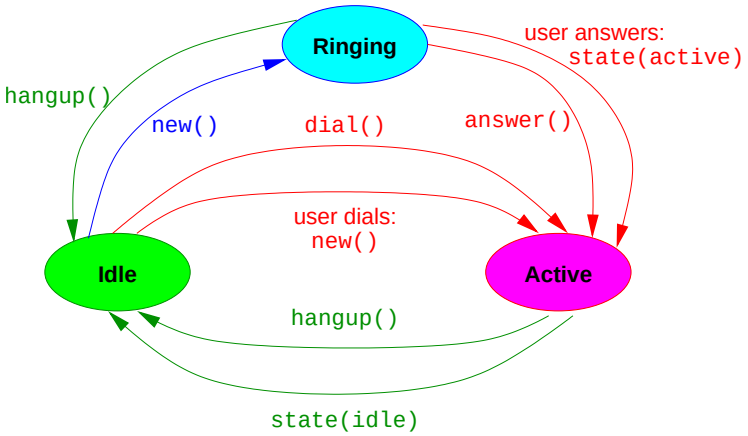
- `const` **`number`** = *own phone number*

Contains the own phone number, usually with country prefix.

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

9.2 Module `phone`: Phone Calls

This module allows to monitor and make voice phone calls. The module can monitor at most one call at the same time. The following diagram depicts the relationship between states and functions:



- If phone.new (p. 200) detects an *incoming call*, this new call is phone.ringing (p. 202). It can either be answered via phone.answer (p. 199) or by the user, or rejected via phone.hangup (p. 200) or by the user. Once the call has been answered, it becomes phone.active (p. 202).
- If phone.new detects an *outgoing call* dialled by the user, or phone.dial (p. 199) successfully establishes one, the call also becomes phone.active.
- An active call can be terminated explicitly via phone.hangup. Alternatively, phone.state (p. 201) can wait for it becoming phone.idle (p. 202), i.e. for its termination.

phone.answer

- function answer()→ null

  Permissions: FreeComm

Answers an incoming (ringing) call by accepting it. This should be called after phone.new (p. 200) returns with an incoming call. See there for an example.

Throws ErrDisconnected if the there is no current call.

phone.dial

- function dial(number, timeout=-1)→ Boolean

  Permissions: FreeComm+CostComm

Dials the given phone number to establish a voice call. If timeout>=0, waits at least timeout milliseconds before giving up. Returns true if the call could be established and the remote party has answered, or false if the timeout was reached.

Throws ErrInUse if a call is already active.

Throws ExcValueOutOfRange if timeout exceeds 2147483 (35 minutes and 47.483 seconds).

```
// make a one minute call to +41797654321
if phone.dial("+41797654321", 30000) then
  sleep(60000);
  phone.hangup()
end
```

phone.hangup

- `function hangup()→ null`

  Permissions: `FreeComm`

Compatibility of function `phone.hangup`
Symbian 3rd Edition phones: a call which is `phone.ringing` cannot be hung up without answering it first. Calling `phone.hangup` on a ringing call will answer it first and then immediately hang up, potentially causing costs for the caller.

Disconnects the current call (''hangs up'' the phone).

Throws `ErrDisconnected` if there is no current call.

Does not hang up a call which was not made via `phone.dial` (p. 199) or obtained via `phone.new` (p. 200).

phone.ms

- `function ms()→ Number`

  Permissions: `FreeComm`

Gets the duration of the current call in milliseconds.

Throws `ErrDisconnected` if there is no current call.

See `phone.state` (p. 201) for an example.

phone.new

- `function new(timeout=-1)→ Array|null`

  Permissions: `FreeComm`

Waits for a new call (incoming or outgoing), and returns an array with

the following fields:

Key	Meaning	Type
incoming	true for incoming, false for outgoing	Boolean
number	Phone number of remote party	String

If timeout>=0, waits at least timeout milliseconds before giving up. Returns null if the timeout was reached.

Throws ExcValueOutOfRange if timeout exceeds 2147483 (35 minutes and 47.483 seconds).

```
// reject all incoming calls from +41797654321
while true do
  c=phone.new();
  if c["incoming"] then
    if c["number"]="+41797654321" then
      // we reject this call
      phone.hangup()
    else
      // other calls are accepted
      phone.answer()
    end
  end
end
```

phone.state

- function state(mask=phone.idle | phone.ringing | phone.active, timeout=-1)→ Number|null

  Permissions: FreeComm

Waits until the current call enters one of the states in mask, and returns the current state. If timeout>=0, waits at least timeout milliseconds before giving up and returning null.

Throws ExcValueOutOfRange if timeout exceeds 2147483 (35 minutes and 47.483 seconds).

Throws ErrDisconnected if there is no current call.

```
// log number and duration of each outgoing call
while true do
  c=phone.new();
  if not c["incoming"] then
    // wait until the call becomes idle again
    phone.state(phone.idle);
    print phone.ms(),"ms call to",c["number"]
  end
end
```

phone Constants

- `const` **`idle`** `= 1` The call is idle, i.e. was hung up.
- `const` **`ringing`** `= 2` A call is coming in and must be answered.
- `const` **`active`** `= 4` A call is active.

10. Applications and Processes

10.1 Module app: Application Control

This module provides access to the applications installed on the phone: listing installed applications, opening documents, starting and stopping applications, and bringing them to the foreground or sending them to the background.

Functions in this module are specific to Symbian OS, and not likely to be portable to other operating systems.

In Symbian OS, each application has its unique UID (unique identifier), which is simply an integer number. In the functions of this module, an application is identified by its UID or its name (caption). Since the caption is language and installation dependent, the UID is generally preferrable. Application UIDs and captions may also vary between different devices.

Since **m** itself is also an application, the functions in this module can also be used to bring **m** to the foreground, send it to background, or simply stop it. The `app.uid` (p. 208) constant identifies the **m** application.

app.find

- `function find(name=null)→ Array`

  Permissions: `ReadApp`

Searches for applications whose name matches the pattern `name`. `name` is not case sensitive and can contain the wildcards `*` and `?`. If `name=null`, searches for all installed applications.

Returns an array with one element for each application found, each element being an array with the following keys:

Key	Meaning	Type
name	Application name (caption)	String
file	Application DLL file name	String
uid	Application UID	Integer

```
// search for the mShell application
for a in app.find("mShell") do
  print a
end
→ [mShell,C:\System\Apps\mShell\mShell.app,270549657]
```

app.hide

- function hide(uidOrName)→ null

  Permissions: ReadApp

Hides the application identified by uidOrName, i.e. sends it to the background. uidOrName can be the application's UID, or its name (caption).

Throws ErrNotFound if the application does not exist.

```
// hide the messaging application
app.hide("Messaging")
```

app.key

- function key(scancodes)→ null

  Permissions: ReadApp+WriteApp

  Capabilities: extended
- function key(keycodes, uidOrName)→ null

  Permissions: ReadApp+WriteApp

  Capabilities: extended

Sends a keyboard event or a series of keyboard events to the device or to a specific application.

With one argument, sends scancodes to the device. scancodes can be a single integer, an array of integers, or a string. A positive integer

causes a press of the key with this scan code, a negative integer a release of the key with this scan code (after changing its sign). Scan codes are OS and device specific. Use ui.cmd (p. 83) after calling ui.keys(true) to obtain the scan code for a specific key.

With two arguments, sends keycodes to the application defined by uidOrName. keycodes can be a single integer, an array of integers, or a string. Each integer or character causes a stroke of the key with this code. Most key codes correspond to character codes, but some codes are reserved for device specific keys. Use ui.cmd (p. 83) after calling ui.keys(false) to obtain the key code for a specific key.

```
// Start the contacts application and send it a name
app.start("Contacts"); app.key("William", "Contacts")
// Simulate flip close and open on UIQ
app.key(0x77); sleep(2000); app.key(0x76)
// Show profile selection via power key on S60
app.key([0xa6, -0xa6])
```

app.open

- function open(file, uidOrName=null)→ Number

  Permissions: Read+Write(file)+ReadApp+WriteApp

Compatibility of function app.open	
Sony Ericsson UIQ3 phones cannot handle uidOrName#null.	ErrNotSupported

Opens a file, using the application defined by uidOrName. uidOrName can be the application's UID, or its name (caption). If uidOrName=null, the standard application for files of this type is used.

Returns the UID of the started application.

Throws ErrNotFound if the application does not exist.

```
// show an image file in the standard image viewer
uid=app.open("mShell.png");
// kill the app after ten seconds
sleep(10000); app.stop(uid)
```

app.runs

- `function runs(uidOrName)→ Boolean`

  Permissions: `ReadApp`

Checks whether the application defined by `uidOrName` is running. `uidOrName` can be the application's UID, or its name (caption).

Throws `ErrNotFound` if the application does not exist.

```
// check whether the phone application is running
// the caption is in german...
app.runs("Telefon")
→ true
```

app.send

- `function send(uidOrName, msgUid, params)→ null`

  Permissions: `ReadApp+WriteApp`

  Capabilities: `extended`

Send a message to the application defined by `uidOrName`. `uidOrName` can be the application's UID, or its name (caption). `msgUid` must be an integer identifying the message type, and `params` must be a string whose bytes define the message.

Throws `ErrNotFound` if the application does not exist or is not running.

This function is completely Symbian OS specific; using it requires additional information typically found in the Symbian OS SDKs. See also `app.view` (p. 208).

```
// have the WML browser open a link
// WML browser has UID 0x10008d39 on Series 60
app.send(0x10008d39, 0, "http://wap.248.ch")
```

app.show

- `function show(uidOrName)→ null`

  Permissions: `ReadApp+WriteApp`

Shows the application identified by uidOrName, i.e. brings it to the foreground. uidOrName can be the application's UID, or its name (caption).

Throws ErrNotFound if the application does not exist or is not running.

```
// make sure the mShell application is shown
app.show(app.uid)
```

app.start

- function start(uidOrName, background=false)→ null

  Permissions: ReadApp+WriteApp

Starts the application identified by uidOrName. uidOrName can be the application's UID, or its name (caption). If background=true, the application is started in the background, otherwise it is brought to the foreground.

Throws ErrNotFound if the application does not exist.

```
// start the WML browser in the background
// WML browser has UID 0x10008d39 on Series 60
app.start(0x10008d39, true)
```

app.stop

- function stop(uidOrName)→ null

  Permissions: ReadApp

Stops (ends) the application identified by uidOrName. uidOrName can be the application's UID, or its name (caption).

Throws ErrNotFound if the application does not exist.

```
// stop the WML browser
// WML browser has UID 0x10008d39 on Series 60
app.stop(0x10008d39)
```

app.view

- `function view(uidOrName, viewUid)→ null`

  Permissions: `ReadApp`

- `function view(uidOrName, viewUid, commandUid, params)→ null`

  Permissions: `ReadApp+WriteApp`

Switches to a view `viewUid` of the application identified by `uidOrName`. `uidOrName` can be the application's UID, or its name (caption).

With four parameters, sends the view the command `commandUid` and the bytes of the string `params`.

Throws `ErrNotFound` if the application does not exist.

This function is completely Symbian OS specific; using it requires additional information typically found in the Symbian OS SDKs.

```
function showcontact(id)
  // build the parameter block
  params=[1]; // EFocusedContactId
  // encode the id as four byte integer
  for i=1 to 4 do
    append(params, id & 0xff); id = id shr 8
  end;
  app.view(0x101f4cce, // Phonebook application UID
           4, // focused view
           0x101f4ccf, // command UID
           char(params)) // params must be string
end

showcontact(114)
```

app Constants

- `const` **`uid`** `= 0x10204299 | 0xa0002f97 | 0xe7e0cab7` The UID of the **m** application.

10.2 Module async: Asynchronous Function Streams

This module provides functions to create and set up streams which turn an asynchronous function call into a read from a stream: when the asynchronous function completes, the value it returns can be obtained from the stream.

By mapping asynchronous function calls into stream reads and waiting for data being available via io.wait (p. 44), a ''parallel'' wait for the completion of multiple asynchronous functions becomes possible, without resorting to multiple processes and pipes (see module proc (p. 213)).

The typical steps in setting up asynchronous function streams are:

1. Create the stream(s) via async.new (p. 212).
2. Call the function(s) via async.call (p. 211), passing the function to be called as a reference.
3. Use io.wait (p. 44) to wait for completion of any of the functions, or for another stream having data to read.
4. Abort the calls as required via async.abort (p. 211) to free up the modules for other calls (see ''Restrictions'' below for details).
5. Depending on the type of stream returned by io.wait, get the function result via async.result (p. 213), or simply read the available data.
6. Call the functions which completed or were aborted again via async.call (p. 211).

   Steps 3 to 6 form an event loop.
7. When they are no longer needed, close the streams with io.close (p. 39). This also aborts any pending calls on the streams.

For instance, a single process can simultaneously wait for an incoming SMS, the user pressing a key, and a bluetooth connection being made:

```
// setup for UI and Bluetooth
ui.keys(false);
service=bt.start("MyService");
// create three streams
s=[async.new(),async.new(),async.new()];
// call the asynchronous functions
async.call(s[0], &sms.receive);
async.call(s[1], &ui.cmd);
async.call(s[2], &bt.accept, service);
while true do
  // wait for any of the functions to complete
  t=io.wait(s);
  // read the function result
  v=async.result(t);
  case t
  in s[0]: // v is an SMS
    ...
    // receive next message
    async.call(s[0], &sms.receive)
  in s[1]: // v is a keycode
    ...
    // get next keycode
    async.call(s[1], &ui.cmd)
  in s[2]: // v is a BT connection
    ...
    // wait for next connection
    async.call(s[2], &bt.accept, service)
  end
end;
// close the streams
for t in s do
  io.close(t)
end
```

Restrictions

There are two important restrictions to observe when using module module `async`:

- Only calls to native functions can be turned into stream reads. As

all low level asynchronous functions are implemented natively, this normally does not pose a problem.

- Native **m** modules are generally not reentrant, so no new function call can be made while a call of the same module is still pending. For instance, you cannot send an SMS via `sms.send` while an `sms.receive` call is still pending. Attempting to call a module with a pending call throws `ExcModuleBusy`.

async.abort

- `function abort(stream)→ null`

Aborts any pending function call on `stream`; does nothing if there is no pending call.

async.call

- `function call(stream, function, ...)→ null`

Calls the function referenced by `function`, passing the remaining parameters, and returns immediately. As soon as `function` returns, a byte becomes available on `stream`, and the function result can be gotten from `stream` via `async.result` (p. 213).

Any timeout on `function` is ignored, whether it is part of the function's module, or a parameter of the call. For functions called via `async.call`, timeouts must be implemented via `io.timeout` (p. 44), and the pending call explicitly aborted via `async.abort`, if required.

If the call immediately terminates with an exception, the exception will appear to be thrown by `async`call. If a call terminates with an exception asynchronously, the exception will be thrown when waiting for or reading data from the stream.

Throws `ExcNotFunction` if `function` is not a function reference. See also section 2.8 (Reference, p. 36).

Throws `ExcNotNative` if `function` is not a reference to a native function.

Throws `ExcModuleBusy` if the function's module already has a pending call.

Throws `ErrInUse` if the stream already has a pending call.

```
// two functions performing essentially the same
function synchronous()
  return phone.state(phone.idle)
end
function asynchronous()
  // create a new stream
  s=async.new();
  // call phone.state(), returning immediately
  async.call(s, &phone.state, phone.idle);
  // wait for phone.state() to complete
  io.read(s, 1);
  // get the result of phone.state()
  state=async.result(s);
  // close the stream
  io.close(s);
  return state
end
```

async.new

- `function new()→ Native Object`

Creates a new asynchronous function stream accepting function calls and returns it. The returned stream can only be used for asynchronous function calls. It cannot be written to, and a single byte can be read after an asynchronous function call completes.

```
// create an asynchronous function stream
stream=async.new()
```

async.pending

- `function pending(stream)→ Boolean`

Returns `true` if `stream` has a pending call. Returns `false` if there is no pending call.

Use `io.avail` to determine whether a result is available.

```
// start a new call if there is none pending,
// and there is no result to be read
if not async.pending(s) and io.avail(s)=0 then
  async.call(s, &sms.receive)
end
```

async.result

- function result(stream)→ anytype

Get the result of the last function called via async.call.

Throws ErrNotReady if there was no call or if it is still pending.

10.3 Module proc: m Processes

This module manages **m** processes (scripts and executables). It can start and stop, and show and hide processes. It also supports a simple inter-process communication (IPC) mechanism via unidirectional named pipes, and an argument string.

Processes are identified by the name of their script or executable (without path and extension). Since shell processes do not have an associated script and thus no name, they cannot be managed from other processes. For instance, the script c:\documents\mShell\BTScanner.m has an associated process with name BTScanner. Process names are not case sensitive.

proc.arg

- function arg()→ String

Get the argument string specified when the process was started via proc.run (p. 217). For processes started manually from the process list or via the autostart feature, proc.arg returns the empty string.

```
// print the command line argument
print proc.arg()
→ hello
```

proc.close

- `function close(name)→ null`
- `function close()→ null`

With one argument, closes the process with the given `name`. Without an argument, closes the process it is called from.

Closing a process also stops it if it is running. If the process is already closed, or there is no such process, the call is ignored.

```
// stop and close the BTScanner process
proc.close("BTScanner")
```

proc.find

- `function find(name="*")→ Array`

Gets a list of all known scripts or executables in the current document folder whose name matches `name`. `name` is not case sensitive and can contain the wildcards `*` (matches any sequence of characters) and `?` (matches any single character).

Throws `ErrNotSupported` when called from standalone applications.

```
// start all processes which end on "Test"
for f in proc.find("*Test") do
  proc.run(f)
end
```

proc.hide

- `function hide(name)→ null`
- `function hide()→ null`

With one argument, hides the process with the given `name`. Without an argument, hides the process it is called from.

In the mShell application, this call simply shows the list of scripts and modules. In standalone applications, this call hides the application.

If the process is not currently shown, this call does nothing.

Throws ErrNotFound if there is no running process with the given name.

```
// hide the current process
proc.hide()
```

proc.pipe

- function pipe(name, create=true, bufsize=256)→ Native Object

Opens or creates a pipe with name name and returns a stream to read from and write to the pipe. The pipe can be opened by other processes using the same name, thus providing a communication channel between **m** processes.

If create=false, the function throws ErrNotFound if the pipe does not already exist.

If created, the pipe will have a buffer of bufsize bytes. The default size is large enough for efficient inter-process communication (IPC): whenever there is not enough room in the pipe buffer, a write to the pipe will block until another process reads from the pipe to free up space.

However, if the same process reads from and writes to the pipe, the buffer must be large enough to hold all data written between reads. This is the only case where larger buffer sizes may be needed.

Once created, a pipe stream is accessed via module io (p. 36):

- io.read, io.readln, and io.readm read data,
- io.write, io.writeln, io.writem, io.print, and io.println write data,
- io.avail gets the number of bytes which can be read without blocking,
- io.wait waits for data which can be read without blocking,
- io.close closes the stream (but not the pipe). The pipe will be deleted when all streams referencing it have been closed.
- io.ces gets and sets the character encoding scheme. As with files, the default is io.raw.

- `io.timeout` sets the timeout for read and write operations.
- `io.flush` sets the auto flush state. If auto flushing is disabled, `io.flush` must be called to make sure all data is written.

With `io.readm` (p. 42) and `io.writem` (p. 46) are ideally suited for pipes, as data is both written and read by **m**.

Only one process can read from the pipe at a given time. Issuing a read with another read pending (from another process) will throw `ErrInUse`.

Up to sixteen processes can write to the pipe at a given time. Issuing a write when sixteen other writes are pending (from other processes) will throw `ErrNotReady`.

Pipes are unidirectional. For bidirectional communication between processes, two pipes (with different names) are required.

The first trivial example just shows how to read from and write to a pipe:

```
// create a pipe stream and write to it
s=proc.pipe("SamplePipe");
io.writeln(s, "Hello world!");
// read from the pipe what was written into it
print io.readln(s)
→ Hello world!
// close the stream; this will also delete the pipe
io.close(s)
```

A more realistic example consists of two processes with two pipes. The first process in script `Reverser` reads a line from pipe `ReverserIn`, and writes the reversed line to pipe `ReverserOut`:

```
function reverse(s)
  c=code(s);
  i=0; j=len(c)-1;
  while i<j do
    h=c[i]; c[i]=c[j]; c[j]=h; i++; j--
  end;
  return char(c)
end

// create (or open) the two pipes
rin=proc.pipe("ReverserIn");
rout=proc.pipe("ReverserOut");
// loop forever reading, reversing and writing
while true do
  io.writeln(rout, reverse(io.readln(rin)))
end
```

We now can use the reverser process:

```
// make sure the reverser runs
proc.run("Reverser");
rin=proc.pipe("ReverserIn");
rout=proc.pipe("ReverserOut");
io.writeln(rin,"Hello world!");
print io.readln(rout)
→ !dlrow olleH
```

proc.run

- `function run(name, arg="")→ null`

Runs (starts) the process with the given `name`, and the argument string `arg`. If a process with this name is already running, the call is ignored.

`name` is always relative to the ''document'' directory of the current process (`system.docdir` (p. 50)). If there is an executable (`.mex` file) with the given name, it will be loaded. Otherwise, the script (`.m` file) will be loaded.

The argument string is accessed via `proc.arg` (p. 213) from the target process.

Throws `ErrNotFound` if there is no executable or script with the given

name.

```
// start the BTScanner process, passing "hello" to it
proc.run("BTScanner", "hello")
```

proc.runs

- `function runs(name)→ Boolean`

Returns `true` if the process with the given `name` is running, and `false` if it is stopped, or there is no such process.

Throws `ErrNotFound` if there is no executable or script with the given name.

```
// stop the BTScanner process
proc.stop("BTScanner");
// it should not be running now
proc.runs("BTScanner")
→ false
```

proc.show

- `function show(name)→ null`
- `function show()→ null`

With one argument, shows the process with the given `name`. Without an argument, shows the process it is called from.

Showing a process shows its console, or any other view it is displaying. If the process is already shown, the call is ignored.

Throws `ErrNotFound` if there is no running process with the given name.

```
// show the current process
proc.show()
```

proc.stop

- `function stop(name)→ null`
- `function stop()→ null`

With one argument, stops the process with the given `name`. Without an argument, stops the process it is called from, i.e. terminates it.

If the process is not running, the call is ignored.

```
// stop the current process
proc.stop()
```

11. Environment

11.1 Module `accel`: Accelerator Measurements

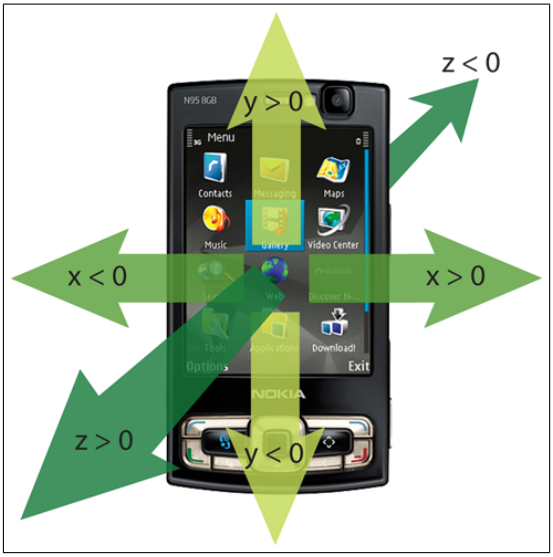
Compatibility of module `accel`	
Sony Ericsson phones and Nokia S60 2nd Edition phones lack this module.	Module not found
Selected Nokia S60 3rd Edition phones devices with builtin acceleration sensor.	ok

This module returns the measurements of the builtin acceleration sensor. Please note that only a few devices have a such a sensor.

Each measurement corresponds to a vector in three dimensional space. Unfortunately the meaning of the values returned varies between devices, in particular the sign of the z component.

On the N95-8GB, we found the following:

Key	Meaning	Type
`x`	Parallel to the screen, negative to the left, positive to the right.	Number
`y`	Parallel to the screen, negative to the bottom, positive to the top.	Number
`z`	Orthogonal to the screen, negative away from the viewer, positive towards the viewer.	Number



Vector directions on the N95-8GB

If the device is still, the acceleration corresponds to the gravitation and thus allows to determine the rotation of the device. For instance, if the device lies flat on a table with the screen facing up, the `z` component will have a large value (gravitation pulling ''down''). If the device is flipped over, the `z` component will change its sign.

The maximum magnitude of the measurements seems to be about 370 for gravitation only. Some devices return higher values for short time acceleration, e.g. when tapping on the device.

accel.get

- `function get()→ Array`

Gets the current acceleration vector.

```
// Check whether the N95-8GB is face up or down
function isFaceUp()
  return accel.get()["z"]<0
end
print isFaceUp()
→ true
```

accel.new

- function new(mindelta=20,timeout=-1)→ Array|null

Waits until any component of the acceleration vector changes by at least mindelta, then return the current vector. If timeout>=0 and timeout milliseconds have passed without any change, null is returned.

The following example draws a blue line in perpendicular direction:

```
// compute the center and radius
s=graph.size();
cx=s[0]/2; cy=s[1]/2; r=cx;
if cy<r then r=cy end;
// compute the scale factor
f=-r/300;
// initially there is no line
dx=0; dy=0;
do
  graph.show();
  a=accel.new(10);
  // erase the old line
  graph.pen(graph.bg());
  graph.line(cx, cy, cx+dx, cy+dy);
  // compute the new line
  dx=f*a["y"]; dy=f*a["x"];
  // draw the new line
  graph.pen(graph.blue);
  graph.line(cx, cy, cx+dx, cy+dy)
  // until the device is flipped
until a["z"]>200
```

Index