Library Part Two

Version 1.06

infowing
style your mobile

m Mobile Shell, Library Part Two, Version 1.06
Written by Lukas Knecht

www.m-shell.net

Document IW-M-LIB2-1.18

Typeset in Switzerland.

Contents

1. Introduction

This manual describes the additional modules available in Part Two of the m standard library. Part Two comprises modules which are:

- highly specialized,
- or are not supported on all devices,
- or are potentially harmful to use,
- or provide significant additional value.

2. Library

2.1 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` (Reference, p. 131)). Valid dates are January 1st, 1980 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` **`todo`** `= 8` Entry is a to-do list item.

Flags can be used to select entries in `agenda.find` (p. 10), and they must be used to indicate the type of the new entry in `agenda.add` (p. 9).

For use in `agenda.find` (p. 10), there is also the value

- `const` **`all`** `= 63` 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` (Reference, p. 131).

```
// 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.

`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.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
```

2.2 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. 16) constant identifies the m application.

app.find

- `function find(name=null) → Array`

  Permissions: `ReadApp`

Searches for applications whose name matches the pattern `name`. `name` 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`
- `function key(keycodes, uidOrName)` → `null`

  Permissions: `ReadApp+WriteApp`

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` (Reference, p. 135) 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` (Reference, p. 135) 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`

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`

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. 16).

```
// 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` The UID of the m application.

2.3 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. 19) creates a new big integer from a number, a string (in a given base, e.g. hexadecimal), or another big integer.
- bigint.num (p. 20) converts a big integer to a number (potentially loosing significant digits).
- bigint.str (p. 21) 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("3333333333333333333333333333333333", -2);
print a, bigint.str(a)
→ bigint@414ffc -6666666666666666666666666666666666
```

bigint.abs

- `function abs(p)→ Native Object`

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

```
r=bigint.abs("-31415926535897932384626 4");
print bigint.str(r)
→ 314159265358979323846264
```

bigint.add

- `function add(p, q)→ Native Object`

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

```
r=bigint.add("12345678901234567890123456789O",
              8765432110);
print bigint.str(r)
→ 1234567890123456789100000000000
```

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("123456789012345678901234567890",
              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("123456789012345678901234567893",
                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("31415926535897932384626 4")
print bigint.str(r)
→ -314159265358979323846264
```

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)
→ 33333333333333333333333333333333
```

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("77155809021290527622553484955867325162 85"+
             "07543313408497691288819319300898474 67");
m=bigint.new("11573371353193579143383022743380098774 49"+
             "56524669244552124759012865929681230709");
c=bigint.pow("36951955703393882182052231534288831920 73"+
             "32988926215558975227889876920636982 3",
             e, m);
print bigint.str(c)
→ 2090963726256956961627254580276511758392932630933805
  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("123456789012345678901234567890", 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("123456789012345678901234567890",
             -8765432110);
print bigint.str(r)
→ 123456789012345678910000000000
```

2.4 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. 30)).

  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. 32).

  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` (Reference, p. 111):

- `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. 30) 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. 32) 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. 29), 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. 26).

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.

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`

Compatibility of function `bt.visible`	
Nokia phones before Symbian 8[a]	ok
Nokia phones with Symbian 8[b]	`ErrNotSupported`
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
```

2.5 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` (Reference, p. 87).

Since the camera is a shared resource and consumes battery power, it must be turned on before use by `cam.on` (p. 36) and turned off afterwards by `cam.off` (p. 36). 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.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.

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

cam.take

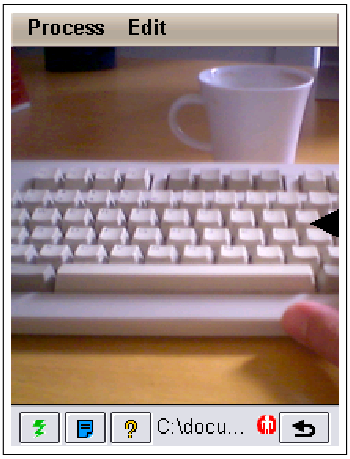
- `function take()→ Native Object`

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

Takes an image of the configured size, brightness and contrast and returns it as an icon (see `graph.icon` (Reference, p. 98)). The icon can be saved, scaled, or analyzed using functions in module `graph` (Reference, p. 87).

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

```
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

cam.view

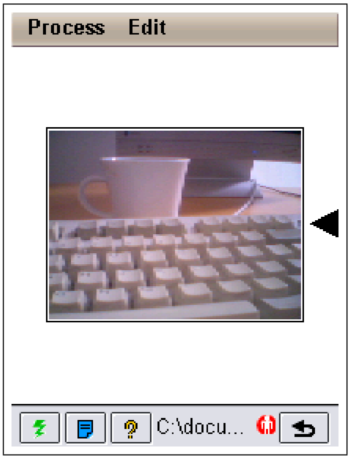
- `function view(x=0,y=0,w=160,h=120)` $\rightarrow$ `Array`

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.

```
// 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

2.6 Module `mms`: Multimedia Messages

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

[a] Only sending of MMS is supported on SE devices.

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` (Reference, p. 125) 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`

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 (Reference, p. 131).
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.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`.

```
// 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. 40) 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
```

2.7 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 not possible. This generally makes little sense anyway, 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` (Reference, p. 111):

- `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.

Internet Access Points

Using TCP/IP requires the phone to connect to an IAP (Internet Access Point), typically via GPRS or UMTS. 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. 48) and `net.stop` (p. 50).

net.adr

- `function adr(hostname)` → `Array`

  Permissions: `CostComm`
- `function adr()` → `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.

Without arguments, returns the local (own) IP address. Getting the local IP address is not supported on all connections and may throw `ErrTimedOut`.

```
print net.adr('www.google.com')
→ [216.239.59.103,216.239.59.104,
   216.239.59.99,216.239.59.147]
print net.adr()
→ [10.122.18.7]
```

`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` (Reference, p. 131).

`serial` and `md5` encode each byte as a string character; use `.code` (Reference, p. 45) 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)→ 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.

Compatibility of Secure Connections	
Sony Ericsson phones	Unreliable, may hang

```
// connect to Infowing's SMTP mail server
s=net.conn("mail.infowing.ch", 25);
// read the prompt
print io.readln(s)
→ 220 mail.infowing.ch Microsoft ESMTP MAIL
  Service, Version: 6.0.3790.1830 ready at  Tue, 24
  May 2005 13:58:44 +0200
// immediately logout again
io.write(s, "QUIT\r\n");
// read the goodbye message
print io.readln(s)
→ 221 2.0.0 mail.infowing.ch Service closing
  transmission channel
// close the connection
io.close(s)
```

For a secure connection example, see `net.cert` (p. 46).

`net.iap`

- `function iap()→ Array`

  Permissions: `CostComm`

- `function iap(setting)→ Array`

  Permissions: `CostComm+WriteApp`

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 index	Number
2	Bearer set supported by this IAP	Number

The preferred IAP index corresponds to an entry in the IAP table in the phone. The bearer set defines the set of bearers supported by this IAP. In most cases, only the "prompt user" flag is of interest, as changing the IAP is rarely required under normal use.

To obtain an IAP index and its bearer set, set the "prompt user" flag to `true`, cause a connection to the IAP (e.g. by resolving a name), then call `iap.net()` (this only works if there is no valid connection to an IAP)

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 to 2, but enable prompting
net.iap([true, 2])
// disable prompting
net.iap(false)
// restore the old setting
net.iap(s)
```

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.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` (Reference, p. 114) also shuts down a connection, but `net.shut` gives finer control over connection termination, and allows to catch errors.

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

net.stop

- `function stop()→ null`

  Permissions: `CostComm`

Stops the current IAP connection. Under normal circumstances, calling this function is not required.

```
// change the IAP, then stop the connection
net.iap([false, 7]);
net.stop()
// obtaining the local IP address should restart
// the connection with the new IAP
net.adr()
```

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.

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.infowing.ch", 25)
  // connection successful...
catch e by
  if index(e, "ErrTimedOut") # 0 then throw e end;
  print "Could not connect within 10 seconds"
end
```

2.8 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. 21).

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. 27) for `obex.uuid` (p. 55).

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`.

A timed out call will always close the OBEX connection; `obex.conn` (p. 52) 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. 52).

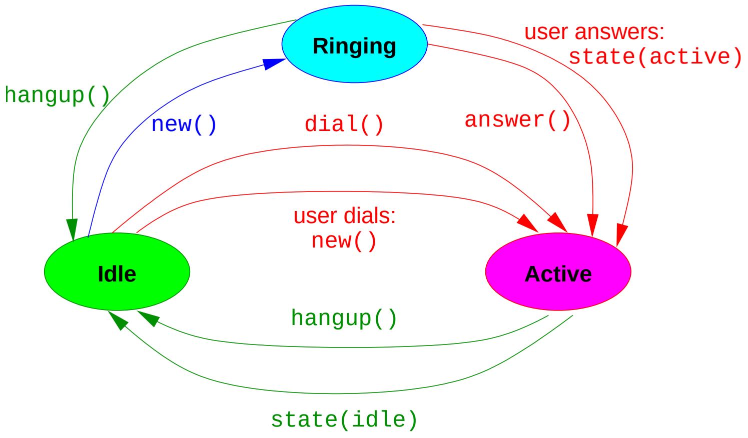
```
// set the "who" name to 'peer1'
obex.who('peer1')
```

obex Constants

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

2.9 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. 58) detects an *incoming call*, this new call is phone.ringing (p. 59). It can either be answered via phone.answer (p. 56) or by the user, or rejected via phone.hangup (p. 57) or by the user. Once the call has been answered, it becomes phone.active (p. 59).
- If phone.new detects an *outgoing call* dialled by the user, or phone.dial (p. 57) successfully establishes one, the call also becomes phone.active.
- An active call can be terminated explicitly via phone.hangup. Alternatively, phone.state (p. 58) can wait for it becoming phone.idle (p. 59), 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. 58) 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.

```
// 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`

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

Throws `ErrDisconnected` if the there is no current call.

Does not hang up a call which was not made via `phone.dial` (p. 57) or obtained via `phone.new` (p. 58).

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. 58) 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.

```
// 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 `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.

2.10 Module `proc`: m Processes

This module manages m processes (scripts). 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. 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. 63). 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, the call is ignored.

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

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

proc.find

- `function find(name="*")→ Array`

Gets a list of all known processes, whose name matches `name`. `name` can contain the wildcards `*` (matches any sequence of characters) and `?` (matches any single character).

```
// 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.

Hiding a process simply shows the standard list of scripts and modules.

If the process is already hidden, the call is ignored.

Throws `ErrNotFound` if there is no 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` (Reference, p. 111):

- `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 (Reference, p. 117) and io.writem (Reference, p. 120) 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 the process is already running, the call is ignored.

The argument string is accessed via `proc.arg` (p. 59) from the target process.

Throws `ErrNotFound` if there is no process 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 closed.

Throws `ErrNotFound` if there is no process 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 was closed, it is opened, and its empty console is shown.

If the process is already shown, the call is ignored.

Throws `ErrNotFound` if there is no 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.

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

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

2.11 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. 65) 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()
```

Index