

Tutorial & User Guide

Version 3.07



m Mobile Shell, Tutorial & User Guide, Version 3.07
Written by Lukas Knecht

www.m-shell.net

Document AB-M-TUT-887

© 2004-2011 airbit AG, 8008 Zürich, Switzerland

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

Typeset in Switzerland.

Contents

1 Introduction **3**
1.1 About **m** 3
1.2 Tutorial Structure 4

2 Quick Start Guide **7**
2.1 Installing **m** 7
2.2 A Sample Script 7
2.3 A Sample Shell Session 11

3 The m Application **13**
3.1 The Script List 13
3.2 The Console 16
3.3 Script Files 19
3.3.1 Sending and Receiving **m** Files 19
3.4 Compiling **m** Scripts 21
3.4.1 Producing Standalone Applications 22
3.5 The Editor 23
3.5.1 Scrolling 25
3.5.2 Find and Replace 26
3.6 The Properties Dialog 28
3.7 The Permissions Dialog 29

4 m Programming **31**
4.1 Basic Arrays 31
4.2 Associative Arrays 35

4.3 Accessing SMS 36
4.4 Editing Data 39
4.5 Making it a Function 43
4.6 Combining SMS and User Interface 44
4.7 Reading and Writing Files 47
4.8 Making it a Module 51
4.9 Conclusion 55

5 m Help System 57
5.1 Invoking help 57
5.2 Navigating through patterns 58

6 m Runtime Errors (Exceptions) 61

7 m Library Overview 65

8 Installation Guide 67
8.1 Installation 67
8.2 Registration 68

Index 69

1. Introduction

This tutorial is a beginner's guide to successfully *writing* **m** *shell scripts*. After working it through, you will know how to operate the **m** application on your phone and have encountered most of the **m** language, including its key functions.

1.1 About m

A honest word first: when we say ''writing an **m** script'', we mean ''programming''. We, the authors of **m**, believe that programming can be a lot of fun. Programming **m** is particularly rewarding. You do it both for and on a device that you often carry with you, maybe wherever you go: first, some clever **m** scripts can make your smart phone a lot smarter; second, you can try new ideas or perfect old ones virtually anytime and anywhere.

So:

- If you are already familiar with any programming language, learning **m** will be easy and straightforward.
- If programming is totally new to you, but you are a curious person interested in technology, **m** scripting can open the door to an exciting and virtually unlimited new activity.
- If however you are totally convinced programming is nothing for you, you should only read sections 2 (p. 7) and 3 (p. 13), which shows you how to use **m** to run scripts written by your friends, or anybody else from the **m** community.

Unless you already are a smartphone Guru, or close to becoming one, learning **m** will also give you a better understanding of the technologies used by your smart phone, like the GSM network, Bluetooth, the agenda or contacts databases, and much more.

Then, a word of warning: smart phones are powerful small computers. Their hardware is in many respects comparable to, and in some respects even superior to, that of a Personal Computer. However, coming out of the factory, smart phones are limited to the capabilities the manufacturer, and often the network provider, have considered worth or safe to include. You can install additional software, but this software will also usually be limited to the tasks it was developed for. There are many good (and a few not so good) reasons this limitation exists. Among the good ones are the following:

- You should not be allowed to manipulate the phone in a way that makes it unusable or leads to loss of important data.
- Cellphone communications are expensive, and often special services will be charged directly to your phone bill. Hence, running the wrong software can become quickly quite expensive.



Unlike a lot of other smart phone software, **m** is very powerful in a general sense. This means that with a malevolent script and you giving the corresponding permissions, **m** can make the phone partially unusable, delete important information, or even charge your phone bill without you noticing it.

This is of course also true for a lot of other software, which you must trust before starting to use it. **m** has a clear advantage here: you can always verify scripts before using them, and you can explicitly deny access to data on your phone or to its communication resources. For the scripts you get from people or sources you do not know or cannot trust, this is highly recommended.

1.2 Tutorial Structure

This tutorial is organized into the following parts:

- A quick start guide to your first **m** steps.
- An introduction to the **m** application and its views.

- A step-by-step tutorial through many aspects of the **m** language. As an example, you build up a more and more complete SMS service.
- An introduction to the **m** help system and coding wizard.
- An overview of the **m** library modules.
- Detailed information on how to install **m**.

This tutorial will not explain every aspect of **m** in full detail. While working through it, and while continuing to work with **m**, you should refer to the following manuals for further and in-depth information:

- **m** Shell Reference (document IW-M-REF) covers all aspects of the **m** language, and some aspects of the **m** application.
- **m** Shell Library (document IW-M-LIB) covers the library of standard **m** modules and functions.

And don't forget: the **m** website at www.m-shell.net is always worth a visit!

2. Quick Start Guide

2.1 Installing m

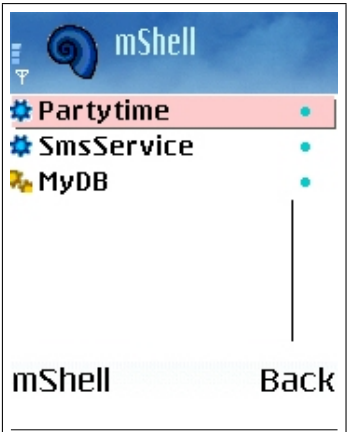
Before you can do your first steps with **m**, you must install it on your phone and optionally register it:

1. Install **m** from the `.sis` file appropriate for your device, e.g. `mShell-S60-3rd.sis` for a S60 3rd Edition phone, or `mShell-UIQ2.sis` for a UIQ2 phone.
2. Start the `mShell` application. It will ask whether you want to register via SMS. If you decide not to register now, you can register later.

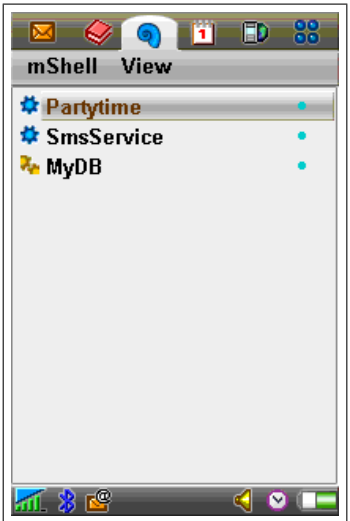
For detailed information about installing and registering **m**, see chapter 8 (p. 67).

2.2 A Sample Script

Once **m** has been activated, it will show you the list of installed scripts and modules.



Series 60 sample screen



UIQ sample screen

A script is like an application you can run. As an example, let's open the `Partytime` script:

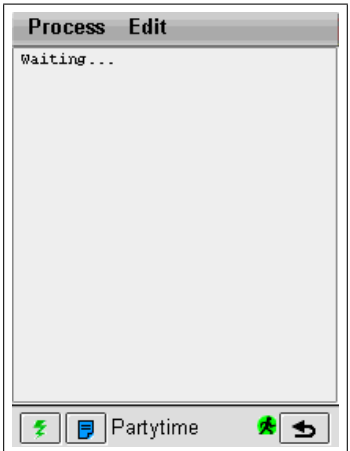
Series 60: Navigate to `Partytime` and press the confirm button.
UIQ: Select `Partytime` with the pen.

This will show the script's empty console. To start the script:

Series 60: Press the confirm button again.
UIQ: Press the ⚡ button.



Series 60 sample screen



UIQ sample screen

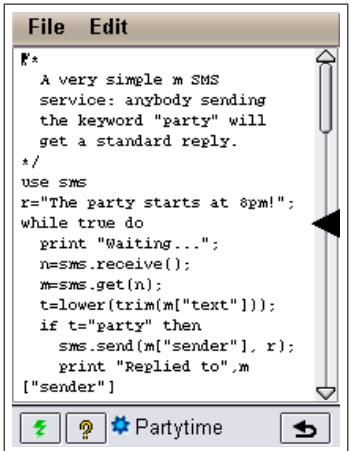
The script runs and prints `Waiting...` on the console. Now have a friend send you an SMS with the text ''party''. After a few seconds she should automatically receive a reply from you, sent from the `Partytime` script. In fact, the script acts as a very simple automatic SMS service.

Stop the script by selecting **Process→Stop**. You have just successfully used your first **m** script!

You can look at the script by selecting **Process→Edit Source**.



Series 60 sample screen



UIQ sample screen

If you want to change the reply message, or the text triggering the reply, simply edit the script. **File→Save & Go** will start the script again, with your changes. See section 3.5 (p. 23) for details.

For your convenience, the complete script is printed here, with comments added:

```
/*
  A very simple m SMS service:
  anybody sending the keyword "party"
  will get a standard reply.
*/
use sms                              // we need it
r="The party starts at 8pm!";        // our standard reply
while true do                        // loop forever
  print "Waiting...";                // let the user know
  n=sms.receive(); m=sms.get(n);     // get the next msg
  t=lower(trim(m["text"]));          // isolate the word
  if t="party" then                  // if it's party,
    sms.send(m["sender"], r);        // send the reply
    print "Replied to",m["sender"] // and log it
  end
end
```

2.3 A Sample Shell Session

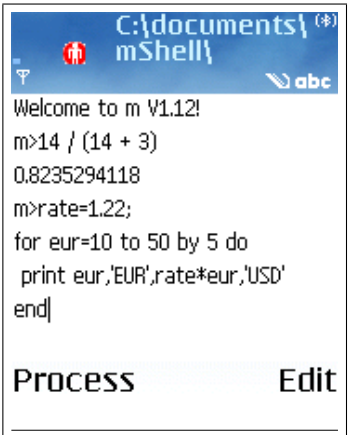
Besides running stored scripts, you can also execute **m** interactively, for instance as a powerful calculator, or to manipulate data on your phone you cannot easily access otherwise.

To open an interactive shell, select **mShell→New Shell**. The shell console will prompt you for a command with `m>`.

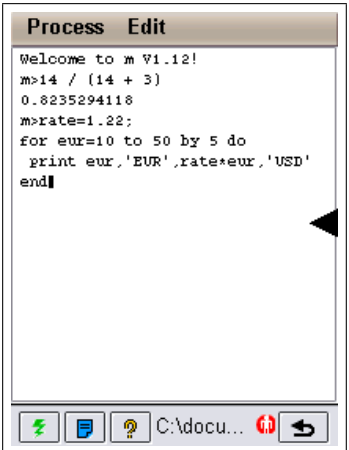
After the prompt, enter the following calculation, then

Series 60: Press the confirm button.
UIQ: Press the ⚡ button.

```
14 / (14 + 3)
→ 0.8235294118
```



Series 60 sample screen



UIQ sample screen

Or print a table of EUR versus USD:

```
rate=1.22;
for eur=10 to 50 by 5 do
  print eur,'EUR',rate*eur,'USD'
end
→ 10 EUR 12.2 USD
   15 EUR 18.3 USD
   20 EUR 24.4 USD
   25 EUR 30.5 USD
   ...
```

Interactive shells can execute arbitrary **m** code, including variable assignments, function declarations, module imports and so on. In addition, there are some useful functions to manipulate files:

`cd`	Display and change current directory.
`cls`	Clear the console output.
`cp`	Copy files.
`del`	Delete files.
`dir`	List directories.
`md`	Create directories.
`mv`	Move files.
`rd`	Remove directories.
`ren`	Rename a file.
`send`	Send a file (''Send As'').
`type`	Display text file contents.

For instance, the following command searches the `C:` drive for JPEG (`.jpg`) files (`/r` indicates that all subdirectories should also be searched):

```
dir c:\*.jpg/r
→ c:\Nokia\Images\Backgrounds\mShellLogo.jpg
   ...
```

See also chapter 3 (Reference, p. 55) for more information about interactive shells.

3. The m Application

All **m** scripts and interactive sessions are run from within the **m** application. The number of scripts which can be run simultaneously is only limited by the phone's resources.

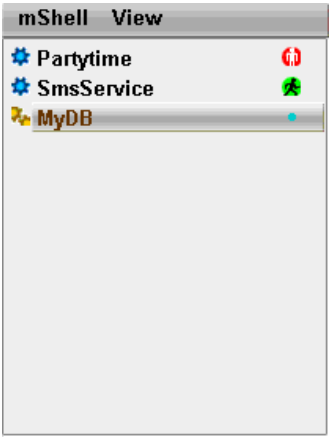
Scripts and their modules can reside in any writable directory on the phone, and scripts and their modules can be organized into folders.

3.1 The Script List

The applications main view shows the list of interactive sessions, and of the folders, scripts, executables, and modules in the current document folder. The current folder can be changed by clicking on a subfolder.



Series 60 sample screen



UIQ sample screen

The script type is indicated by the icon on the left:

 An interactive shell session.

 A subfolder. The ''`..`'' folder indicates the parent folder.

 A subfolder which is part of the Library Path, i.e. is searched for **m** modules. See also section 2.9 (Reference, p. 39).

 A script.

 A compiled `.mex` file which can be executed directly, i.e. an ''executable''. See also section 3.4 (p. 21).

 A module for use by scripts, shells or other modules.

 A script which is started when the **m** application is launched with this current directory.

 An executable which is started when the **m** application is launched.

For a script, executable or interactive session, the icon on the right indicates its state:

 Inactive, without console.

 Stopped, but with a console. **Process**→**Close** will make it inactive.

 Running. **Process**→**Stop** will stop it.

 Waiting for console input.

 Was running, but produced an error. Open it to see the error message.

The script list offers the following menu options:

mShell→New Shell	Create a new interactive shell session.
mShell→New Script	Create a new **m** script.
mShell→New Module	Create a new **m** module.
mShell→New Folder	Create a new subfolder in the current folder.
mShell→Library Path	Add the selected folder to or remove it from the Library Path. Adding it will have **m** search the folder for modules. See also section 2.9 (Reference, p. 39).
mShell→Send As	Send the selected script or module (e.g. via Bluetooth or as an e-mail attachment).
mShell→Delete	Delete the selected script, module or folder, after asking for confirmation.
mShell→Back	Send the **m** application to the background.
mShell→Exit	Exit the **m** application.
View→Properties	Edit the application properties (see 3.6 (p. 28)).
View→Supervisor	Edit the supervisor properties (see 2.3 (Supervisor, p. 9)). This option is only available if the **m** Supervisor & Viewer is installed.
View→Permissions	Edit the permissions granted to **m** scripts (see 3.7 (p. 29)).
View→About	Display a message with details about the **m** application, including the serial number.
View→Toggle Size	(*S60 only*): toggle the view size, showing or hiding the title pane.
View→Run Registration	Register **m** via SMS.

Pressing the delete key will also delete the selected item (after asking for confirmation).

Searching Scripts

On *S60 only*, the script list supports searching via a popup window, very much like searching in the standard contacts application.

Typing a character opens the window and shows only the scripts and modules starting with the typed characters. Typing the clear/backspace key removes the popup window.

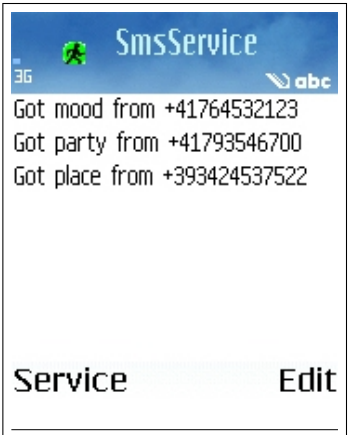


Searching a script on S60

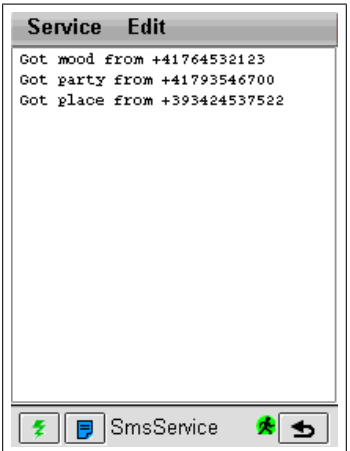
3.2 The Console

Every active script and shell session has a console associated with it. It is displayed if a script or session is selected from the script list.

The console is a simple text viewer displaying the output of the `print` statement (and the `io.stdout` (Library, p. 44) stream). In interactive shells, it also serves to input the **m** statements to be executed.



Series 60 sample screen



UIQ sample screen

To start or continue execution,

Series 60: Press the confirm button.
UIQ: Press the button.

Console text is frozen up to the point of last output. In interactive shells, this is typically all text up to the last `m>` prompt.

The console has a command history. A certain number of previous inputs can be recalled:

Series 60: Press the down key.
UIQ: Press the button, or use **Process**→**Edit Source**.

The console will cycle through the previous inputs.

The console offers the following menu options:

Process→Go	Run the script or command.
Process→Stop	Stop a currently executing script or command.
Process→Close	Close this script's console, or close the shell session.
Process→Edit Source	Edit the source associated with the script, or recall the previous input.
Process→Compile	Compile the script into a `.mex` file which can be executed directly; see also section 3.4 (p. 21).
Process→Auto Go	Toggle the ''auto go'' state of the script or executable. If ''auto go'' is enabled (indicated by a  or  icon), the script or executable is started automatically whenever the **m** application starts. Note that scripts are only started if they reside in the current directory. Executables are always started.
Process→Save Output	Save the console text to a file, using the current source file encoding.
Process→Clear Output	Clear the console, i.e. remove all text.
Process→View Size	(*S60 only*): toggle the view size, showing or hiding the title pane.
Edit→Back	(*S60 only*) return to the script list.
Edit→Help	Show the help for the text before the cursor (see chapter 5 (p. 57)).
Edit→Error Trace	Show the error traceback for the last runtime error (see chapter 6 (p. 61)).
Edit→Copy	Copy the selected text to the clipboard.
Edit→Cut	Cut the selected text to the clipboard.
Edit→Paste	Paste text from the clipboard.
Edit→Find	Open a find/replace dialog and start find mode (see section 3.5.2 (p. 26)).

3.3 Script Files

m scripts and modules are just files stored on the file system of your phone. Since they are files, you can manipulate them using a file explorer application or using **m**, and transfer them to other devices.

These files are located in the document directory of the **m** application. This directory can be changed from within the application by navigating between folders (i.e., subdirectories).

Scripts have the file extension `.m`, modules the extension `.mm`, and compiled executables the extension `.mex`. For instance, the `Partytime` script might correspond to file `c:\documents\mShell\Partytime.m`. You can easily verify this from within a shell: open a shell (e.g. with **mShell→New Shell**) and try the following (you don't have to enter the comments starting with `//`):

```
// get the document directory from module system
m>use system system.docdir
→ c:\documents\mShell\
// show the file contents
type(system.docdir+"Partytime.m")
→ /*
     A very simple m SMS
     service: anybody sending
     the keyword "party" will
     get a standard reply.
   */
   use sms
   r="The party starts at 8pm!";
   ...
```

3.3.1 Sending and Receiving m Files

Since **m** scripts and modules are ordinary files, you can send them to other phones or a PC, and you can also receive them. The easiest way to transfer files is usually via Bluetooth, but other transport media like USB cable, Internet or MMS are also possible.

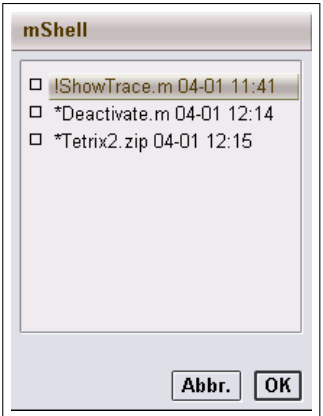
- To *send a file*, either use **mShell→Send As** from the script list, or

File→Save & Send As from the editor.

- To *receive a file*, simply have another device send it to you such that it appears in your inbox. Then run the `inbox2m` script. It scans your inbox for `.m`, `.mm`, `.mex` and `.zip` files, then shows the available files in a list. Select the ones you want to load into **m**.



Series 60 sample screen



UIQ sample screen

Since `inbox2m` supports `.zip` files and extracts them into the document directory, you can install entire packages with several scripts, modules and supporting files in one go.

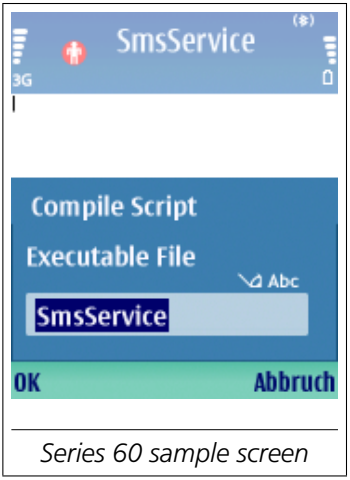
3.4 Compiling m Scripts

An **m** script and the modules it requires can be compiled into a single ''executable'' file. These files have the extension `.mex` and are ideal to quickly distribute **m** applications in a single file among **m** users.

A `.mex` file is marked in the script list by a  icon. It can be executed directly from the **m** script list by clicking on it.

Creating a `.mex` file is also the first step in creating an installable application.

As an example, open the `SmsService` script from the script list, then choose **Process→Compile**. A dialog will appear asking you for the name of the executable file (without `.mex` extension).



Series 60 sample screen

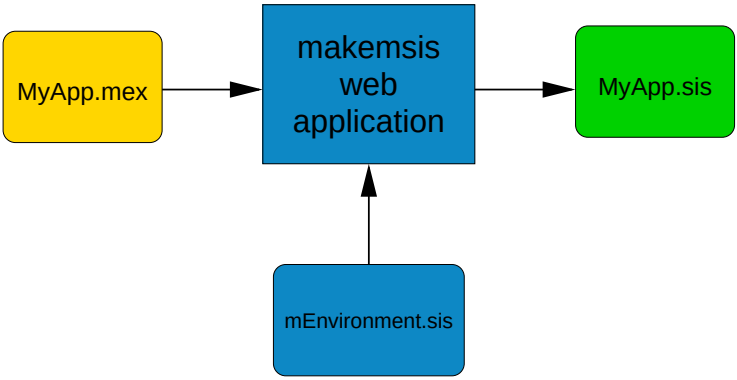


UIQ sample screen

After successful compilation, a new item `SmsService` with icon  will appear once you switch back to the script list. This will contain everything that's needed to run `SmsService` from within **m**.

3.4.1 Producing Standalone Applications

Standalone applications for Symbian OS can easily be produced from applications written in **m**. These can be installed on other devices, without requiring previous installation of **m**. Simply upload your `.mex` file to a web application located on www.m-shell.net/Makemsis.aspx, select the platform you want the `.sis` created for (S60 2nd, 3rd or 5th edition, UIQ2 or UIQ3), and have the web application create a `.sis` file you can download and install on other devices.



Your `.mex` file will be combined with the **m** runtime environment `mEnvironment.sis` for the selected platform.

Here are the essential steps to create an `SmsService.sis` from `SmsService.mex`:

1. Copy `SmsService.mex` to a PC with internet connection, for instance by using **mShell→Send As**.
2. On the PC, go to www.m-shell.net/Makemsis.aspx, and upload `SmsService.mex` as **.mex File**, clicking on **Add Files**.
3. In **Settings**, select the appropriate platform, e.g. ''S60 3rd'', then click on **Create SIS**.
4. In **Your Files**, you will find `SmsService.sis` to download. This can now be installed on other devices.



Installed `SmsService` *application*

The web application has several options, for instance to change the application title or caption, to supply another icon, or to install additional files. These are explained in chapter 4 (Reference, p. 61).

3.5 The Editor

To write or edit an **m** script or module, you can use the editor:
To edit an existing script, open it from the script list, then

Series 60: Choose **Process**→**Edit Source**.
UIQ: Press the button.

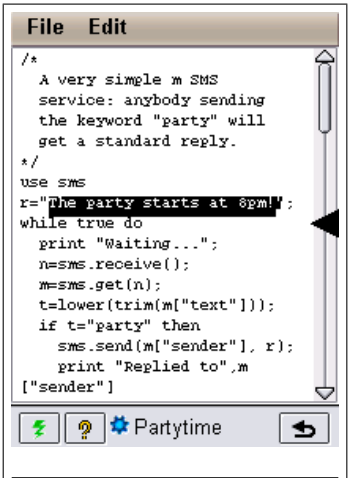
To edit an existing module, simply open it from the script list.

Only one file can be edited at a time: when loading a file, a previously edited file will be saved.

As an example, open the `Partytime` script and load it into the editor, as explained above.



Series 60 sample screen



UIQ sample screen

Now replace the text ''The party starts at 8pm!'' by ''Sorry, no party tonight!''.

You can start the modified script directly from the editor by

Series 60: choosing **File→Save & Go**
UIQ: pressing the ⚡ button.

Note that on *S60*, the confirm button inserts a new line when in the editor.

The editor offers the following menu options:

File→Save & Go	Save the file and run the script.
File→Save & Compile	Save the file and compile the script.
File→Save As	Save the file (optionally changing the name), but stay in the editor.
File→Save & Close	Save the file, and return to the script list.
File→Save & Send As	Save the file, then send it (e.g. via Bluetooth or as an e-mail attachment).
File→Discard & Close	Discard any changes, and return to the script list.
File→Delete	Discard any changes, and delete the file.
File→Rename	Rename the file.
File→View Size	(*S60 only*): toggle the view size, showing or hiding the title pane.

The **Edit** menu is the same as in the console.

3.5.1 Scrolling

On *S60*, clicking the the shift (select) button once enters *pagewise mode*: the up and down keys scroll backwards and forward one page at a time. Pressing any other key or clicking the shift button again returns to linewise mode.

On *UIQ*, the editor offers a vertical scroll bar to quickly move to another part of the source text.

3.5.2 Find and Replace

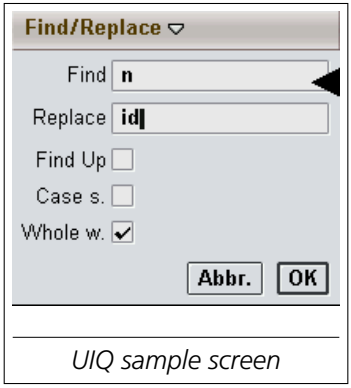
The editor has a simple find/replace mechanism: **Edit→Find** opens a dialog where find/replace arguments can be entered. The arguments are:

`Find`	The text to search for.
`Replace`	The text to replace the search text with.
`Find Up`	Select whether the initial search direction is up (backwards) or down (forward).
`Case s.`	Select whether the search is case sensitive.
`Whole w.`	Select whether the search only considers whole word matches, ignoring substrings.

To replace all occurences of variable `n` by variable `id` in the `Partytime` script, choose **Edit→Find** in the editor, then enter the search arguments. Select `Whole w.`, as we only want to find isolated instances of the letter `n`, not `n` as part of another word.



Series 60 sample screen



UIQ sample screen

Press **Ok** to find the first occurence of `n`. After starting the search, finding the first occurence and selecting it, the editor enters *find mode*, indicated by the 🔍 icon. In find mode, three keys have a special meaning:

S60 Key	UIQ Key	Meaning
Arrow Up	Jog Dial Up	Search backwards for the next occurrence of the find string.
Arrow Down	Jog Dial Down	Search forward for the next occurrence of the find string.
Confirm	Jog Dial Press	Replace this occurrence, then search for the next occurrence of the find string in the same direction as before.

All other keys leave find mode. Find mode is also left if the find string does not occur in the given direction.

Hence, to replace the next occurence of `n` by `id` and search the next occurence,

Series 60: press the confirm button.
UIQ: press the Jog Dial.



Series 60 sample screen



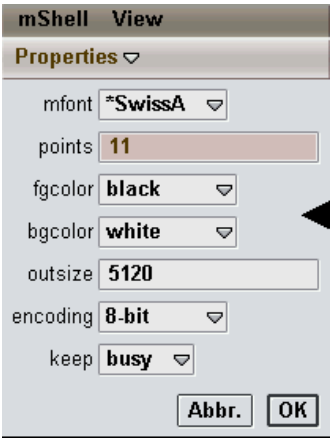
UIQ sample screen

3.6 The Properties Dialog

The properties dialog is accessed via **View→Properties**. The application properties determine visual properties of **m**, and its behaviour with respect to the system.



Series 60 sample screen



UIQ sample screen

The individual properties are:

`mfont`	The **m** console font. A font with a leading asterisk is scalable.
`points`	The size of the font in pixels. This is ignored if the font is not scalable.
`fgcolor`	The foreground (text) color of console and editor.
`bgcolor`	The background color of console and editor.
`outsize`	Size of console buffer in characters.
`encoding`	The encoding for source files: typical choices are `bom` and `utf-8`; other options are `utf-16le`, `utf-16be` and `8-bit`. See section A.3 (Reference, p. 78) for details.
`keep`	Select whether system exit requests are ignored; if set to `busy`, requests are ignored if any script is running.
`phone`	Your own phone number, to be assigned to `gsm.number` (Library, p. 246).

See appendix A.3 (Reference, p. 78) for details about properties.

3.7 The Permissions Dialog

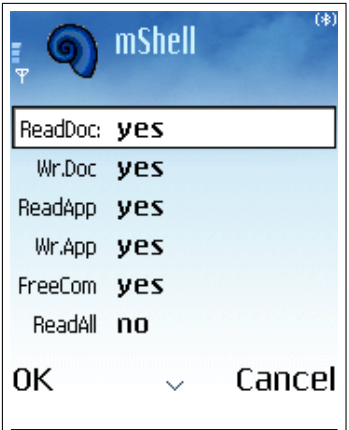
The permissions dialog is accessed via **View→Permissions**. These permissions grant or deny access to data or resources on the phone.

There are three data areas which can be protected individually from reading and/or writing:

1. `Doc`: the directory where **m** scripts and modules reside, and all its subdirectories and files. Granting access to the `Doc` area is normally safe, as it cannot harm the phone, only the **m** scripts.
2. `App`: the data modifiable through well defined interfaces, like contacts, the agenda, messages and such. Granting read access to the `App` area is normally safe, as it cannot harm the phone. When write access to the `App` area is granted, a malevolent **m** script can destroy valuable data.
3. `All`: any directory or file on the phone. Granting read access to the `All` area is normally safe, as it cannot harm the phone. However, granting write access to the `All` area is generally not recommended, as a malevolent script can render the phone unusable. It also allows a script to indirectly grant any other permission, e.g. `CostComm`.

In addition, two communication areas can also be protected:

1. `FreeComm`: any communication which is free, i.e. where charges cannot occur. This includes reading messages and Bluetooth.
2. `CostComm`: any communication which may be subject to charges. This includes sending messages and networking via TCP/IP.



Series 60 sample screen



UIQ sample screen

The individual permissions are:

`ReadDoc`	Grant read access to the `Doc` area.
`WriteDoc`	Grant write access to the `Doc` area.
`ReadApp`	Grant read access to the `App` area.
`WriteApp`	Grant write access to the `App` area.
`FreeComm`	Grant access to free communication resources.
`ReadAll`	Grant read access to the `All` area.
`WriteAll`	Grant write access to the `All` area.
`CostComm`	Grant access to chargeable communication resources.
`Device`	Grant write access to the device state (power, reboot).

See also appendix 1.2 (Library, p. 6) for details about permissions.

4. m Programming

After having explored the **m** application, you are ready to start writing **m** scripts. This chapter will introduce you to the **m** language and some of its functions. At the end, you will have seen how a keyword driven SMS service can be implemented, including a persistent database for its content, and a user interface to edit it.

In particular, you will see how to:

- build a small database of keywords and responses using **m** arrays,
- monitor incoming SMS traffic,
- send SMS responses on incoming messages matching a keyword from the database,
- implement a user interface to edit the database,
- combine SMS monitoring and user interface with a menu,
- save and load the database from and to a file,
- move the database part to its own module, so it is reusable from other scripts.

The resulting **m** components, the script `SmsService` and the module `MyDB`, are part of the standard installation.

4.1 Basic Arrays

Our SMS service should examine each incoming SMS and check whether it matches a list of keywords we defined. If a match is found, the corresponding reply should be sent back. Let's assume we initially start with the following keywords and replies:

Keyword	Reply
party	The party starts at 8pm!
place	I am at home.
mood	Just don't ask.

For instance, if someone sends you an SMS with the text ''mood'', your phone should automatically reply ''Just don't ask.''.

In **m**, we could represent this table as two *arrays*:

```
keywords=["party", "place", "mood"];
replies=["The party starts at 8pm!",
         "I am at home.",
         "Just don't ask."]
```

An array is a collection of values (numbers, strings, other arrays...). The above two statements create two arrays and assign them to the *variables* `keywords` and `replies`.

A few observations may help clarifying:

- A *variable* is just a name we can assign a value to. In **m**, names are case sensitive, so `keywords` and `KeyWords` are two different names. Blanks or interpunction cannot be used in names, and they must not start with digits.
- The `=` operator assigns a value to a variable.
- Two assignments (and two statements in general) must be separated by a semicolon (`;`).
- An array is defined by a comma separated list of values between brackets (`[]`).
- A string must be quoted. Both single quotes (`'`) or double quotes (`"`) can be used.

Single elements of each array can be accessed by *indexing* :

```
print keywords[0]
→ party
print replies[2]
→ Just don't ask
print replies[3]
→ ExcIndexOutOfRange thrown
print len(replies) // The number of elements in replies
→ 3
```

And again a few remarks:

- Indexing happens by appending the element index between brackets after the array variable.
- The index of the first element is zero[1].
- Using an index number for which no element exists is an error: it throws `ExcIndexOutOfRange`. See section 2.10 (Reference, p. 42) for more information about exceptions.
- The number of elements (length) of an array can be obtained by calling the `len` function on the array.
- Functions are called by their name (e.g. `len`), followed by the arguments between parentheses `()`.
- The rest of the line after two slashes (`//`) is considered a comment and ignored by **m**.

Now that we have `keywords` and `replies` defined, how are we going to use them? Remember we want to find the reply for an incoming message. This means we have to search through all keywords. If we find a match, the corresponding reply can be used. In **m**, we could write something like this:

[1] Consequently, the index of the last element is the number of elements minus one. This may appear strange, but follows most modern programming languages. Starting indexing at zero has proven much easier to deal with in practice than starting at one.

```
msg=...; // the incoming message
i=0; // start at the first element
while i<len(keywords) and keywords[i]#msg do
  i++
end;
if i<len(keywords) then
  reply=replies[i];
  // send the reply
end
```

The above code fragment introduces two very important **m** *control structures*, `while` and `if`:

- `i=0` assigns zero to the variable `i`.
- The expression between `while` and `do` is evaluated. If it is true, the statements between `do` and `end` are executed.
- `i<len(keywords)` checks whether `i` has not yet reached the end of the array.
- `keywords[i]#msg` checks whether `keywords[i]` is *not* equal to `msg`.
- If both conditions are true, we move to the next element: `i++` simply adds one to `i`. We could also have written `i=i+1` instead.
- The `while` loop ends if either the end of the array has been reached, or `keywords[i]` equals `msg`.
- The expression between `if` and `then` is evaluated. If it is true, the statments between `then` and `end` are executed:
- If `i<len(keywords)`, `keywords[i]` must equal `msg`, and we reply with the corresponding text `replies[i]`.

As an example, consider `msg="place"`. With `i=0`, the `while` condition is true, so `i++` is executed, setting `i=1`. Since `keywords[i]` now equals `msg`, the `while` condition is no longer true. And since `i<len(keywords)`, the reply `replies[i]` will be sent.

4.2 Associative Arrays

In the previous section, we solved the problem of looking up data which matches a given key in a table. Since this problem occurs very frequently in programming, **m** offers a standard solution for it: *associative arrays*, or arrays whose elements can be accessed directly with (string) keys. If we define our tiny database of keywords and replies in a single array as follows:

```
db=["party": "The party starts at 8pm!",
    "place": "I am at home.",
    "mood": "Just don't ask."];
```

we can access the elements directly by their keys:

```
print db["place"]
→ I am at home.
print db["mood"]
→ Just don't ask.
print db["hello"]
→ null
```

A few remarks:

- An associative array is constructed by prefixing each array element with a key and a colon (`:`). The key must be a string.
- A key in brackets (`[]`) directly indexes into the table and accesses the corresponding element.
- Using an index string for which no element exists is not an error, but returns the special value `null`. This is in contrast to indexing with numbers, where the corresponding element must exist.

An associative array is still a normal array and can also be indexed by numbers:

```
print len(db)
→ 3
print db[1]
→ I am at home
```

Array elements can also be modified via their keys, and new elements can be added by indexing with a new key:

```
db["place"]="I am at work.";
db["hello"]="How do you do?";
print len(db)
→ 4
print db
→ ["The party starts at 8pm!","I am at work.",
   "Just don't ask.","How do you do?"]
```

4.3 Accessing SMS

Given our small database `db`, we can now look into receiving and sending SMS.

To access the SMS functionality of your phone from **m**, we load the corresponding *module*. There are many modules for different functions of your phone, or of **m**; chapter 7 (p. 65) gives you an overview. A few good reasons for having modules:

- **m** can be extended by new modules, adding to its power and flexibility. This includes modules written by yourself, as you will see later on.
- Isolating different concepts into separate modules clarifies **m** and makes it easier to understand and learn.
- Only loading a module when it is needed saves memory.

The module giving SMS access is called, not surprisingly, module `sms` (Library, p. 202). Using it, our service could look as follows:

```
// load the SMS module
use sms
// define the reply database
db=["party": "The party starts at 8pm!",
    "place": "I am at home.",
    "mood": "Just don't ask."];
while true do
  // wait for a message
  id=sms.receive();
  // get the message
  msg=sms.get(id);
  // get the trimmed text in lowercase
  t=lower(trim(msg["text"]));
  // if we find it in our database, reply
  if db[t]#null then
    print "Got",t,"from",msg["sender"];
    sms.send(msg["sender"], db[t]);
    sms.delete(id)
  end
end
```

A few explanations:

- A module is loaded (or imported) with the `use` command.
- The loop `while true` will forever execute the code between `do` and the corresponding `end`. That's exactly what we want: our service should only stop if we stop the process from the **m** application.
- Functions from a module are called by their name, prefixed by the module name and a dot: `sms.receive()` calls function `receive` from module sms.
- `sms.receive()` waits until a new SMS arrives, and returns a number identifying the new message. We assign this number to variable `id`.
- Note the empty argument list after `sms.receive`. These are required to make it clear to **m** that we want to call a function.
- `sms.get(id)` retrieves the message with the id we got from `sms.receive()`, and returns it as an associative array. The array has, among others, the following members:

`sender`	The phone number of the sender of the message.
`text`	The text of the message.

As you see, associative arrays are also often used by the **m** library, whenever a set of related values has to be dealt with.

- Our service should also work if the message sent contains leading or trailing blanks, and case should not matter. We therefore use two functions built into **m**: `trim` to remove blanks, and `lower` to convert all uppercase characters to lowercase. All put together, we can simply write `lower(trim(msg["text"]))`.
- We test whether our database in variable `db` contains a reply for the message text `t`. Remember that `db[t]` returns `null` if there is no element for key `t`, so if `db[t]` does not equal `null`, we have a reply.
- If there is a reply in the database, a confirmation is printed on the console, and the reply is sent with `sms.send()`. This function takes two arguments: the recipient of the message, and the message text. Since we send a reply, the recipient is the sender of the original message, which we find in `msg["sender"]`.
- After sending the reply, we delete the message, as we do not want our SMS inbox to fill up with messages we already replied to. `sms.delete(id)` deletes the message with the id we got from `sms.receive()`.

When creating automated replying systems, it is a good idea to check whether we are not accidentally replying to a message coming from ourselves, and entering into a never ending loop. Let's assume we have added a keyword `"echo"` with reply `"echo"`:

```
db["echo"]="echo";
```

If we now send ourselves a message ''echo'', the service will start to reply to itself until it is stopped.

This is unlikely to happen, but if it does, the consequences may be quite expensive.

In **m**, there is a simple way to check whether we sent a message to ourselves: if **m** is properly configured, `gsm.number` (Library, p. 246)

contains our own phone number which we can check against. The above check for a valid message can be completed by:

```
// if it's not from us, and we find it
// in our database, reply
if msg["sender"]#gsm.number and db[t]#null then
```

Don't forget to add module `gsm` (Library, p. 243) to the `use` clause:

```
use sms, gsm
```

4.4 Editing Data

At this point we have a working SMS service. However, to modify replies or add new keywords, we must modify our script. This is not really all that user friendly. It would be much better to have a graphical user interface allowing us to edit the database. We would like to be able to modify replies for existing keywords, and to add new keywords and replies.

In **m**, it is fairly easy to construct such a user interface using the functions from module module `ui` (Library, p. 101):

First, the user should be able to choose one of the existing keywords to modify it, or pick an item `<New>` if she wants to add a new keyword/reply pair:

```
use ui, array

list=keys(db);
array.sort(list);
array.insert(list, 0, "<New>");
i=ui.list(list);
```

This code fragment, if executed on our database variable `db`, shows the following dialog:



Series 60 sample screen



UIQ sample screen

Comments:

- We need two other modules, module `ui` (Library, p. 101) and module `array` (Library, p. 22).
- The builtin `keys` function is called to obtain an array with the keys from `db`. We assign it to variable `list`.
- The `array.sort()` function sorts the list alphabetically.
- The `array.insert()` function inserts the string `<New>` at the beginning of the list.
- Eventually, the `ui.list()` function is called to display the dialog. This function returns when the user chooses an item, or cancels the dialog. We assign the result to variable `i`.

Once the user has made her choice, we can add a new key-value pair, or edit an existing one.

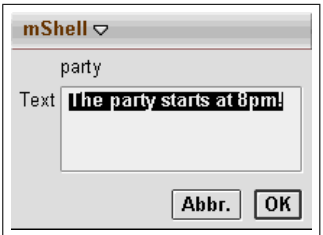
```
i=i[0];
if i=0 then // <New> was selected: add
  f=ui.form(["Key":"","Text":"\n"]);
  if f#null then
    k=f["Key"];
    if db[k]=null then db[k]=f["Text"]
    else ui.error(k + " already exists") end
  end
else // an existing keyword was selected: edit
  k=list[i];
  f=ui.form([k,"Text":db[k]+"\n"]);
  if f#null then db[k]=f["Text"] end
end
```

A few explanations:

- If the user did not cancel the dialog, `ui.list` returns an array with the indices of the selected items. Since the call to `ui.list` did not specify multiple items to be selected, the index array `i` will always have a single index `i[0]`.
- If the selected index is zero (`i=0`), the user has chosen `<New>` (since `list[0]="<New>"`), and we display a dialog to add a keyword and a reply.
  Otherwise, the reply of the keyword at `list[i]` is to be edited, and we display the corresponding dialog.
- The `ui.form()` function takes an associative array of values to be edited, and displays a corresponding dialog. The keys of the array become labels in the dialog:



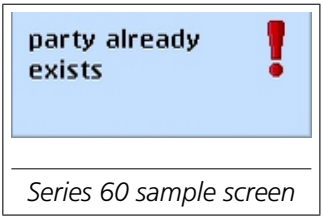
Series 60 sample screen



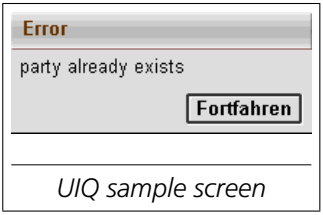
UIQ sample screen

Inside the strings for the `Text` fields, you will note a `\n`. This is the code for a line break (the `n` stands for ''newline''). Having a new line in the contents for a `ui.form()` field marks this field as multi-line, so the field can contain several lines, and can also scroll vertically.

- `ui.form` returns `null` if the dialog has been canceled. If this happens, we do nothing. Otherwise, `f` is an associative array containing the edited values: for instance, `f["Text"]` contains the edited reply text.
- Before adding a new pair with keyword `k`, we check whether it already exists. If it does, we display an error message with `ui.error()`:



Series 60 sample screen



UIQ sample screen

4.5 Making it a Function

We now have the bits and pieces together to create a *function* in **m** which edits any array of key-value pairs, for instance our `db` variable. We would like to have a function `edit`, which we can simply call, passing our database as a parameter:

```
edit(db)
```

And here is such a function:

```
function edit(table)
  while true do
    // display the list of keywords
    list=keys(table);
    array.sort(list);
    array.insert(list, 0, "<New>");
    i=ui.list(list);
    // if the user canceled, i is null
    if i=null then break end;
    i=i[0];
    if i=0 then // <New> was selected: add
      f=ui.form(["Key":"","Text":"\n"]);
      if f#null then
        k=f["Key"];
        if table[k]=null then table[k]=f["Text"]
        else ui.error(k + " already exists") end
      end
    else // an existing keyword was selected: edit
      k=list[i];
      f=ui.form([k,"Text":table[k]+"\n"]);
      if f#null then table[k]=f["Text"] end
    end
  end
end
```

- A function is defined by the keyword `function`, followed by its name, and the argument list in parentheses `()`. Here, there is a single argument, `table`, which is the array we want to edit (if there are multiple arguments, separate them by commas).

- The following code up to the corresponding `end` is the body of the function, which will be executed each time it is called.
- All variables inside the function, including the arguments, are *local* to the function; they are not the same variables as those outside the function. For instance, the statement

  ```
  list=keys(table)
  ```

  only modifies the variable `list` in the function, not any other variable with this name used outside the function or in another function.
- The whole editing process is put into a loop, which is repeated until the user cancels the list dialog. If this happens, `i=null`, and the `break` statement is executed, leaving the loop and eventually returning from the function.

So if we write

```
edit(db)
```

this means executing the function `edit`, passing our variable `db` to it. During the call, `table=db`, and all modifications to the elements of `table` are in fact modifications to the elements of `db`.

For an in-depth presentation of functions, refer to section 2.8 (Reference, p. 32).

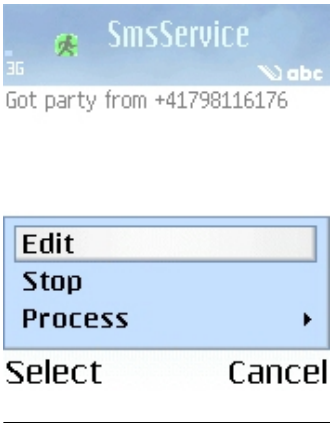
4.6 Combining SMS and User Interface

Now that we have our user interface, we would like to allow the user to edit the database while the SMS code presented in section 4.3 (p. 36) is running.

Doing so is relatively straightforward by adding a menu with two options: the `Edit` option should run the editor (our function `edit()`), and `Stop` should stop the script. With function `ui.menu()`, it is trivial to install a menu:

```
ui.menu("Service",["Edit","Stop"])
```

adds a menu with title `Service` and the two options:



Series 60 sample screen



UIQ sample screen

If the user picks an option, `ui.cmd()` will return it:

```
print ui.cmd()
→ Edit
```

But now we have a problem: if the user hasn't picked an option before, `ui.cmd()` will wait. Likewise, `sms.receive()` will wait until an SMS arrives. So we have two events to wait for, but we can only wait for one at a given point in our code.

There is a simple solution to this: both `sms.receive()` and `ui.cmd()` take a timeout: they do not necessarily wait forever, but optionally only for a certain period. Almost all functions in **m** which wait for a certain event have such timeouts. The timeout period is always indicated in milliseconds (ms, 1/1000 of a second). If the timeout expires, the functions typically return `null`.

For instance,

```
sms.receive(1000)
```

waits one second for a new message, then simply returns `null` if no

message arrives within this period[2].

With this simple method, we can combine the user interface and the SMS monitoring[3]:

```
ui.menu("Service",["Edit","Stop"]);
do
  id=sms.receive(1000);
  if id#null then // there is a new message
    msg=sms.get(id);
    t=lower(trim(msg["text"]));
    if db[t]#null then
      print "Got",t,"from",msg["sender"];
      sms.send(msg["sender"], db[t]);
      sms.delete(id)
    end
  end;
  cmd=ui.cmd(5000);
  if cmd="Edit" then
    edit(db)
  end
until cmd="Stop"
```

Remarks:

- The `do-until` loop executes code until a condition becomes true: in this case, until the user picks `Stop` from the menu. It is similar to the `while-do-end` loop, but the condition is tested at the end of the loop.
- If `sms.receive()` times out, it returns `null`: no message can be checked in this case.
- The script does not respond to a pick from the menu while `sms.receive()` is executing. To minimize the time this happens,

[2] Or has arrived before `sms.receive()` was called.

[3] This technique of regularly polling two inputs is not ideal for a cellphone. Even if no message is arriving and we are not editing our database, the **m** application wakes up every few seconds to check for either event. This unnecessarily drains the battery. Better solutions are possible, e.g. with interrupting `ui.menu` or with multiprocessing, but are beyond the scope of this tutorial.

the timeout for the `sms.receive()` is only one second, whereas the timeout for `ui.cmd()` is five seconds.

- If `ui.cmd()` times out, the variable `cmd` becomes `null`, which is different from both `"Edit"` and `"Stop"`. There is no need to check this case explicitly.

4.7 Reading and Writing Files

So far, we have an SMS service which automatically responds to incoming messages, and allows the keywords and messages to be edited. But our script is not perfect: whenever it stops, all changes to the database are lost. We must make our database *persistent*, so it is still around when we restart the script, even after turning off the phone.

To persistently save data, our phone offers a *file system*. This is very similar to file systems on other computers, be it Windows® or a UNIX®-like system. The main difference is that by far the most common media to store files on larger computers are hard disks, whereas your phone most likely uses memory chips, but this doesn't matter at all. The idea of the file system remains the same.

As on Windows, the file system is organized into drives with directories (folders) and subdirectores. Each file has a name which must be unique to its directory. Section 1.3 (Library, p. 7) tells you more about it.

To access a file from **m**, module module `io` (Library, p. 43) is used. Using this module, a function `save()` saving our database to a file could look as follows:

```
use io
function save(table, file="table.dat")
  f=io.create(file);
  for k in keys(table) do
    io.writeln(f, k);
    io.writeln(f, table[k])
  end;
  io.close(f)
end
```

Some explanations:

- `save()` takes two arguments, the `table` to save, and a `file` name. The file name is optional and defaults to `"table.dat"`. So the two following calls are equivalent:

```
save(db);
save(db, "table.dat")
```

  A function can have as many optional arguments as needed, provided they are the last ones. Section 2.8 (Reference, p. 34) gives you the details.

- `io.create(file)` creates a new, empty file and returns a handle to it. This handle can then be used to write to and read from the file. The handle is assigned to variable `f`.

- The loop starting `for k in keys(table) do` is executed once for each element in the array returned by `keys(table)`.

- `io.writeln(f, k)` writes a line with the string `k` (the keyword in our database) to the file represented by `f`. The contents of the table (the reply) is written to the next line.

- After all keywords and replies have been written, `io.close(f)` closes the file.

If we execute the following code:

```
save(db)
```

the file `table.dat` may contain this (use a shell session to easily type the contents of a file):

```
m>type table.dat
→ party
   The party starts at 8pm!
   place
   I am at home.
   mood
   Just don't ask.
```

A function `load()` to read this data back in could look as follows:

```
function load(file="table.dat")
  table=[];
  try
    f=io.open(file);
    k=io.readln(f);
    while k#null do
      table[k]=io.readln(f);
      k=io.readln(f)
    end;
    io.close(f)
  catch e by end;
  return table
end
```

This function is slightly more complicated:

- We start with an empty array, and assign it to local variable `table`.
- `io.open(file)` opens an existing file to read it and, like `io.create()`, returns a handle to it. However, if the file doesn't exist, it throws `ErrNotFound`. To cope with this case, the call to `io.open()` is put into a `try-catch-end` block: within such a block, any exception thrown will be catched, and execution continues with the statements between `catch` and `end`. Here, there are no such statements, so `table` remains empty if `io.open()` fails. This is exactly what we want.
- If `io.open()` is successful, we read the first line from the file using `io.readln(f)`, which must be a keyword. `io.readln()` returns `null` if there is no more data, so we can use a `while` loop to go through all keywords.
- Inside the `while` loop, the next line is assigned to `table[k]`, appending an element with the key `k` to our table, followed by reading the next keyword.
- After all lines have been read, the file is closed with a call to `io.close()`.
- `return table` returns the contents of variable `table` as the function result. To load our database from default file `table.dat`, we simply call `load()` and assign its result to our variable `db`:

```
db=load()
```

The two functions `load()` and `save()` we have presented above do their job nicely. But there is a small problem: our reader assumes the both keywords and replies fit on exactly one line. However, in our `edit()` function, we explicitly allowed replies to cover multiple lines.

To solve this problem, we could add a special separator token marking the end of a line, making `load()` considerably more complicated. But **m** offers a much simpler solution: the two functions `io.readm()` and `io.writem()` allow to write (almost) any **m** value directly to a file, and read it back in, all in one go. `io.writem()` not only writes the data, but also information about its type, the length of arrays, their keys, etc. `io.readm()` uses this information to reconstruct the value from the file[4].

The disadvantage is that the file written is no longer a simple text file you can edit yourself. Instead, it is a binary file highly sensitive to changes, so it is best to treat such files as a black box.

With these two functions, saving and loading becomes particularly easy:

```
use io
function save(table, file="table.dat")
  f=io.create(file);
  io.writem(f, table);
  io.close(f)
end

function load(file="table.dat")
  try
    f=io.open(file);
    table=io.readm(f);
    io.close(f);
    return table
  catch e by
    return []
  end
end
```

- `save()` now just creates the file, writes `table`, and closes the file

[4] In computer jargon, this is called *serialization* and *deserialization*.

again.

- `load()` does exactly the opposite, and returns the data read. If the file does not exist or cannot be read, this is catched by the `try-catch` block, and we return an empty array: `return []`.

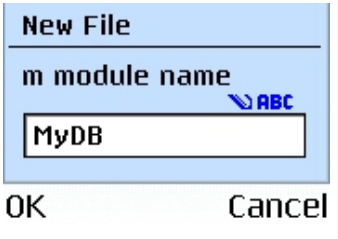
4.8 Making it a Module

Before we add all the pieces together, we want to introduce a last concept: modularization. Remember we have written three functions:

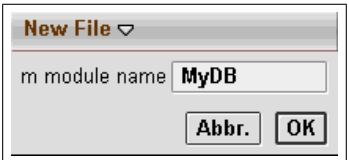
- `function edit(table)` to edit a database table,
- `function save(table, file)` to save a database table to a file.
- `function load(file)` to load a database table from a file.

These three functions can manage any database of key-text pairs, not just the keywords and replies for our SMS service. We should therefore make them generally available, and create a module from them.

We will call our module `MyDB`. The module is part of the standard installation, so you don't have to create it with **mShell→New Module**:



Series 60 sample screen



UIQ sample screen

Instead, you can simply open `MyDb` to look at the module code below:

```
/**
   A simple key-data database.
*/
use io, array, ui

/**
   Load the database from a file.
   @param file the file to load the database from.
   @return the database.
*/
function load(file="table.dat")
  try
    f=io.open(file);
    table=io.readm(f);
    io.close(f);
    return table
  catch e by
    return []
  end
end
```

```
/**
   Save the database to a file.
   @param table the database to save.
   @param file the file to save the database to.
*/
function save(table, file="table.dat")
  f=io.create(file);
  io.writem(f, table);
  io.close(f)
end
```

```
/**
   Present a user interface to edit a database.
   @param table the database to edit.
*/
function edit(table)
  while true do
    list=keys(table);
    array.sort(list);
    array.insert(list, 0, "<New>");
    i=ui.list(list);
    if i=null then break end;
    i=i[0];
    if i=0 then
      f=ui.form(["Key":"","Text":"\n"]);
      if f#null then
        k=f["Key"];
        if table[k]=null then table[k]=f["Text"]
        else ui.error(k + " already exists") end
      end
    else
      k=list[i];
      f=ui.form([k,"Text":table[k]+"\n"]);
      if f#null then table[k]=f["Text"] end
    end
  end
end
```

Remarks:

- The module source is, like a script source, just a sequence of **m** `use` clauses, function definitions and statements. Outside the module, the functions and variables will be accessible by prefixing them with the module name, e.g. `MyDB.load`.
- Since a module will be read by others who are trying to understand what it has offer, it is a good idea to add comments. Multi-line comments start with slash-star (`/*`), and end with star-slash (`*/`). All characters in between are ignored by m. We recommend the tags `@param` and `@return` known from Java™ to comment on parameters and return values.

To finish, we have a look at SmsService, which uses MyDB:

```
/**
   A configurable SMS service.
*/
use sms, mydb, ui

const file="SmsService.dat";
db=mydb.load(file);
ui.menu("Service",["Edit","Stop"]);
do
  id=sms.receive(1000);
  if id#null then
    msg=sms.get(id);
    t=lower(trim(msg["text"]));
    if db[t]#null then
      print "Got",t,"from",msg["sender"];
      sms.send(msg["sender"], db[t]);
      sms.delete(id)
    end
  end;
  cmd=ui.cmd(5000);
  if cmd="Edit" then
    mydb.edit(db); mydb.save(db, file)
  end
until cmd="Stop"
```

Remarks:

- The `mydb` in the `use` list makes sure the `MyDB` module is loaded. Note that, unlike variables and functions, module names are *not* case sensitive. This is because names in the Symbian OS file system are not save sensitive, so the modules `MyDB` and `mydb` cannot be distinguished.
- The database is written to and read from file `SmsService.dat`. We assign this to variable `file` and make it `const`, so it cannot be modified. This is not really necessary; it is mainly a hint to the human reader.
- Before the service starts receiving SMS, we load the database by

calling a function from our module: `mydb.load(file)`.

- Every time the database has been edited, it is saved: `mydb.edit()` is immediately followed by `mydb.save()`.

And that's all there is!

4.9 Conclusion

Hopefully, this chapter has presented enough of **m** to get you started. A good point to continue would be to further extend `SmsService`. For instance, you could:

- make content only available to numbers found in your contacts database, (see `contacts.findnr` (Library, p. 145)),
- add variables to the content, for instance to include information about your location (see `gsm.cid` (Library, p. 243)),
- add other information to the content, for instance whether the incoming messages should be deleted, or a count for the number of messages received,
- play a certain sound if a certain message arrives (see `audio.play` (Library, p. 212)).

Or you simply start experimenting towards the perfectly smart phone you always dreamt of: the next chapter presents a tour d'horizon of the **m** library of functions to give you some ideas of what''s possible and what you could achieve.

5. m Help System

The **m** help system adds IDE (Interactive Development Environment) functionality to the editor. It offers source context sensitive information about **m** language constructs, its library, and other **m** modules, and significantly reduces the typing required to enter correct **m** code.

5.1 Invoking help

The help system is invoked from the editor (or the interactive shell):

Series 60: Click the ''yes'' (green phone call) button.
UIQ: Press the [?] button.

Alternatively, you can invoke it with **Edit→Help**.

The help being displayed depends on the code before the cursor position:

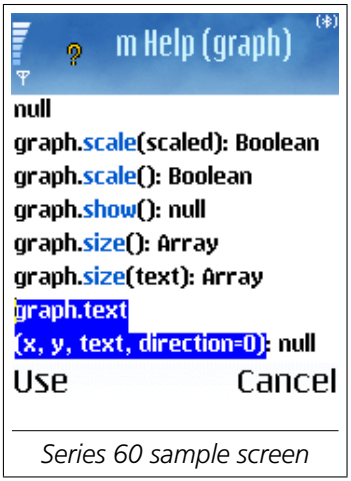
- If it is a function or constant from a module, the help for this module is displayed, with the corresponding function or constant selected. Module aliases (`use ... as ...`) are resolved before looking up the module.
- If it is a keyword or a builtin function or constant, the default help is displayed, with the corresponding language construct or constant selected.

Let's assume the following code fragment:

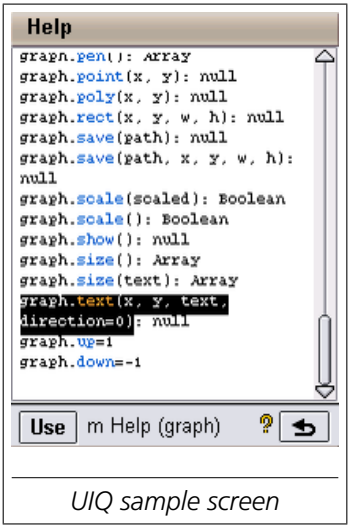
```
use graph as g

for i=1 to 10 do
  g.t
end
```

If the cursor is positioned just after `"g.t"` (i.e. you have just typed it in) and you invoke help, the following page will be displayed:



Series 60 sample screen



UIQ sample screen

Note that the first function or constant matching the code before the cursor is selected. In our example, this is `graph.text`.

You can select another function by navigating up or down.

5.2 Navigating through patterns

If you select an item from the help by pressing **Use** or the confirm key, it is inserted into the code. Its variable parts (arguments) are then quoted between « and » (''french quotes'').

If there are such arguments at or after the cursor, pressing the confirm key in the editor (Jog Dial on UIQ) gets a different meaning: it selects the entire argument, thus allowing to:

- simply replace it by characters you type,
- remove it by pressing the delete or backspace key,
- retain it without quotes by pressing the confirm key again.

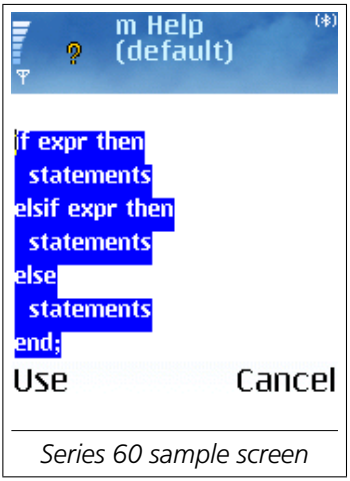
This feature significantly reduces the number of keystrokes required to enter code: let's assume you have continued writing the code fragment

to draw text, and now want to add an `if-then-else` construct to color the text, alternating between red and green:

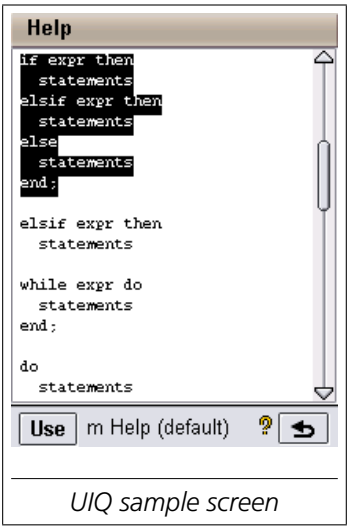
```
use graph as g

for i=1 to 10 do
  if
  g.text(10, 15*i, "i="+i)
end
```

Invoking help with the cursor after `"if"` will display the following help screen:

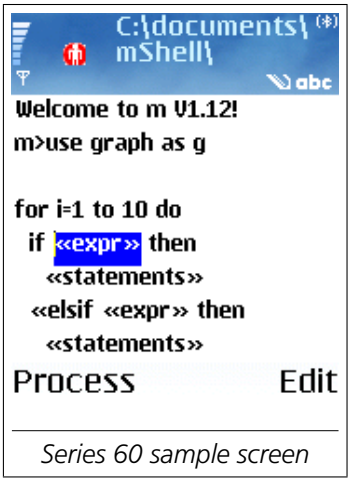


Series 60 sample screen

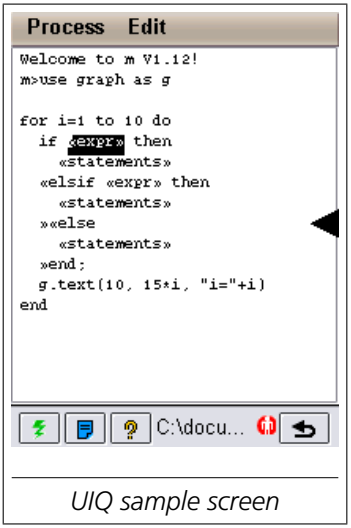


UIQ sample screen

Pressing **Use** inserts this skeleton into the code. Note that the first argument «expr» is already selected, so you can immediately replace it: type `i%2=1`, then press confirm to select «statements». Replace it by typing `g.pen(g.red)`, maybe using help again.



Series 60 sample screen



UIQ sample screen

Pressing confirm again selects the entire `"elsif"` clause. We don't need it, so press the delete key.

Pressing confirm again selects the entire `"else"` clause. We want it to select a different color if `i%2#1`, so press confirm to just remove the quotes. This also selects `«statements»`. Replace it by `g.pen(g.green)`.

The final code now looks as follows:

```
use graph as g

for i=1 to 10 do
  if i%2=1 then
    g.pen(g.red)
  else
    g.pen(g.green)
  end;
  g.text(10, 15*i, "i="+i)
end
```

It is a good idea to practice a little bit with the help system, in particular with the argument selection feature. Also, browsing through the default help gives you an overview of the language constructs supported by help.

6. m Runtime Errors (Exceptions)

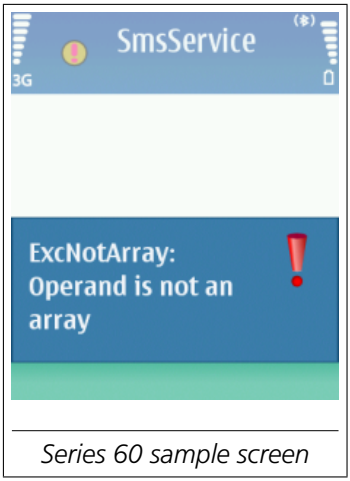
Uncatched exceptions in **m** scripts will terminate the script, and the exception will be shown as an error message, followed by a list of the nested function calls leading to the error. In a script, selecting a function call will open the corresponding source and show the location of the call or statement leading to the exception.

In an executable, the source locations are not available.

To demonstrate the handling of uncatched exceptions, let's assume we forgot in our SMS service to test for the user cancelling the dialog with the choice of messages to edit, i.e. the `ui.list()` call returning `null`. Open the `MyDB` module, and comment out the line checking for returned value being `null` (by putting two slashes in front of it):

```
    i=ui.list(list);
    //if i=null then break end;
    i=i[0];
```

Now save the modified module, and start `SmsService`. After choosing **Edit** from the menu, cancel the selection list. The following message should appear:



Series 60 sample screen



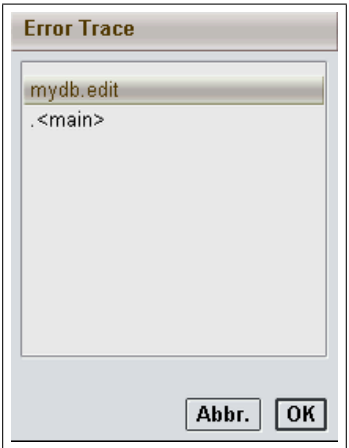
UIQ sample screen

The exception `ExcNotArray` indicates that an expression (or variable) which was not an array was accessed as an array, but it was no array (we know it was `null`).

Confirming the message dialog shows the list of nested function calls which produced the error. The topmost is the function where the error occured, the last is normally `.<main>`, i.e. the statements of the main part of the main script (the ''module'' with empty name).

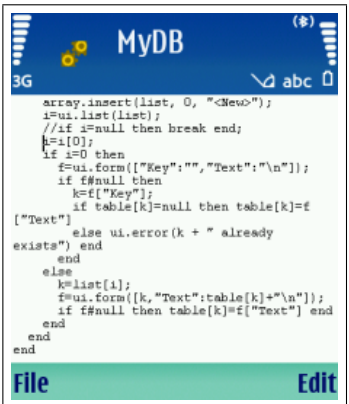


Series 60 sample screen

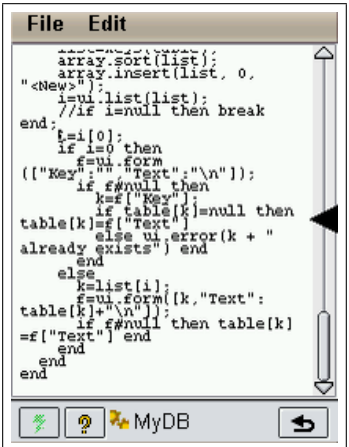


UIQ sample screen

Select `mydb.edit`. This will show the corresponding module source, with the cursor conveniently placed at the location of the error[1]:



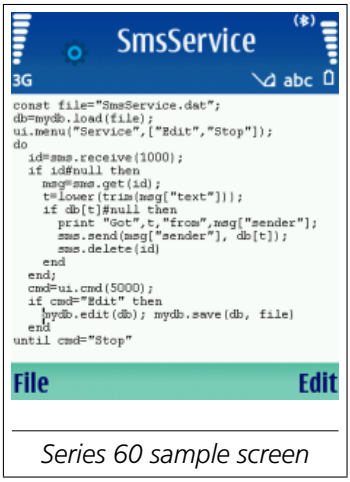
Series 60 sample screen



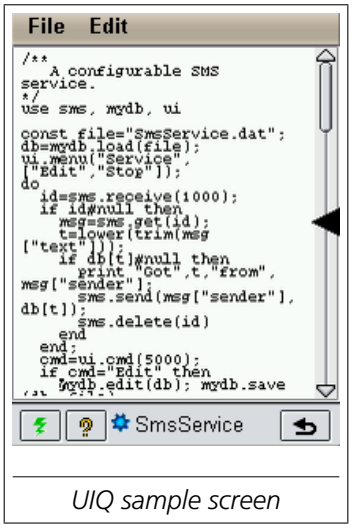
UIQ sample screen

[1] On UIQ3 devices, the cursor is unfortunately not always placed correctly.

Selecting **Edit**→**Error Trace** will display the last error trace again, so you can pick `.<main>` to show the location where `mydb.edit` was called:



Series 60 sample screen



UIQ sample screen

Following the traceback should help you getting a better understanding of the conditions which lead to the error.

Note that in standalone applications, uncatched runtime errors will simply show a message, and terminate the application.

7. m Library Overview

m comes with roughly 170 basic functions, organized into modules.

These modules give access to the different components of the phone and its operating system, or simply add support for the **m** language:

- builtin functions (Library, p. 9): The builtin functions for type conversion, string and array handling, comparison, and type tests.
- module `array` (Library, p. 22): Array functions
- module `audio` (Library, p. 207): Audio functions
- module `contacts` (Library, p. 141): Contacts database
- module `digest` (Library, p. 281): Message digests
- module `encoding` (Library, p. 29): Standard encodings
- module `files` (Library, p. 34): File and directory access
- module `graph` (Library, p. 73): Screen graphics
- module `gsm` (Library, p. 243): GSM information
- module `io` (Library, p. 43): File and stream input/output
- module `math` (Library, p. 128): Mathematical functions
- module `memio` (Library, p. 58): In-Memory Streams and Buffers
- module `sms` (Library, p. 202): Short messages
- module `system` (Library, p. 61): System related functions
- module `time` (Library, p. 65): Time and date functions
- module `ui` (Library, p. 101): User interface functions

In addition to the above modules, there are modules which are more specialized, offering hundreds of additional functions. However, they are not supported equally well on all devices, or missing from some devices altogether.

- module `accel` (Library, p. 269): Accelerator Measurements
- module `agenda` (Library, p. 133): Agenda Database
- module `app` (Library, p. 251): Application Control
- module `async` (Library, p. 257): Asynchronous Function Streams
- module `bigint` (Library, p. 123): Large Integers
- module `bt` (Library, p. 151): Bluetooth Communication
- module `cam` (Library, p. 215): Onboard Camera
- module `comm` (Library, p. 163): Serial Communications
- module `compass` (Library, p. 271): Magnetometer Measurements
- module `light` (Library, p. 98): Device Lighting
- module `location` (Library, p. 273): Location Information
- module `mail` (Library, p. 179): E-Mail Messages
- module `mms` (Library, p. 189): Multimedia Messages
- module `msg` (Library, p. 195): Generic Message Access
- module `net` (Library, p. 167): TCP/IP Networking
- module `obex` (Library, p. 199): Object Exchange Client
- module `phone` (Library, p. 246): Phone Calls
- module `proc` (Library, p. 262): **m** Processes
- module `vibra` (Library, p. 121): Vibration Control
- module `video` (Library, p. 229): Playing and Recording Videos
- module `zip` (Library, p. 69): Unpacking ZIP files

8. Installation Guide

8.1 Installation

Like any other Symbian OS application, **m** is installed from a `.sis` (Symbian Installation System) file.

Get the most recent version of the installation file for your device. You can always download the most recent files from www.m-shell.net, the official **m** website, together with accompanying documentation. For Windows®, there are complete installers. For other operating systems, download the `.zip` file.

Currently, there are four supported device types. For the two Symbian 2nd Edition device types, there are just two installation files:

S60 2nd edition	mShell-S60-2nd.sis
UIQ2	mShell-UIQ2.sis

For Symbian 3rd/5th Edition devices (Symbian 9 and higher), there are three variants due to platform security:

	Self signed	Online signable	DevCert signable
S60 3rd	mShell-S60-3rd.sis	mEnvironment-S60-3rd-OS.sis, mShell-S60-3rd-OS.sis	mEnvironment-S60-3rd-DC.sis, mShell-S60-3rd-DC.sis
S60 5th	mShell-S60-5th.sis	mEnvironment-S60-5th-OS.sis, mShell-S60-5th-OS.sis	mEnvironment-S60-5th-DC.sis, mShell-S60-5th-DC.sis
UIQ3	mShell-UIQ3.sis	mEnvironment-UIQ3-OS.sis, mShell-UIQ3-OS.sis	mEnvironment-UIQ3-DC.sis, mShell-UIQ3-DC.sis

The three variants are (see also 5.2 (Reference, p. 68) for details):

- The ''Self signed'' package is ready to install, but with some **m** functions not permitted by Symbian Platform Security. It contains the **m** environment as an embedded package.
- The ''Online signable'' packages must be signed online on www.symbiansigned.com and installed separately (`mEnvironment` first). After installation, executing **m** functions requiring extended capabilities will also be permitted.

- The ''DevCert signable'' packages must be signed with your own Symbian Developer Certificate and installed separately (`mEnvironment` first). After installation, executing **m** functions requiring certified capabilities will also be permitted.

On Windows, the simplest way to install **m** is:

1. Click on `Symbian Files` in the `m Mobile Shell` start menu.
2. Right click on the appropriate install file and choose **Send To→Bluetooth**.
3. Follow the instructions on screen.

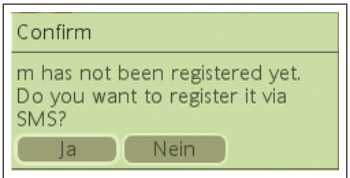
You can install **m** on any storage device you like, either the built-in memory or the removable memory card.

8.2 Registration

Once installed, you should register **m** via SMS. Registration serves a twofold purpose: it registers you as an **m** user, and it helps us to get reliable statistics about **m** usage.



Series 60 sample screen



UIQ sample screen

During registration, **m** sends an SMS to the registration server. If you decide not to register immediately, you can run the SMS registration process any time by executing the **View→Run Registration** command.

Index

.mex, 14, 18, 21, 22
.sis, 22, 67
.zip, 20

application, 13
array, 31
    associative, 35
    indexing, 32
array.insert, 40
array.sort, 40
assignment, 32
associative array, 35
auto go, 18

bgcolor, property, 28

Case s., 26
color, 28
command history, 17
compiling, 21
console, 16
    color, 28
    font, 28
control structures, 34
CostComm, user permission, 30

Delete, 15
deserialization, 50
Device, user permission, 30
document directory, 19

edit, 43, 53
editor, 23
encoding, property, 28
ErrNotFound, 49
exception, 61
ExcIndexOutOfRange, 33
ExcNotArray, 62
executable, 14

fgcolor, property, 28
file, 47
    module, 19
    script, 19
file extension, 19
file system, 47
Find, 26
find
    editor, 26
find mode, 26
Find Up, 26
folder, 13, 14
font, 28
FreeComm, user permission, 30
function, 43

help system, 57

IDE, 57
if, 34
inbox2m, 20

Installation, 67
installation file, 67
io.close, 48, 49
io.create, 48
io.open, 49
io.readln, 49
io.readm, 50
io.writeln, 48
io.writem, 50

keep, property, 28
keys, 40

len, 33
Library Path, 14, 15
load, 48, 52
lower, 38

MEX file, 22
mfont, property, 28
module, 36
    file, 19
MyDB module, 52

New Folder, 15
New Module, 15
New Script, 15
New Shell, 15

outsize, property, 28

pagewise mode, 25
Partytime, 8
permissions, 29
phone, property, 28
points, property, 28
properties, 28

ReadAll, user permission, 30
ReadApp, user permission, 30
ReadDoc, user permission, 30
receive a file, 20
Registration, 68
Replace, 26
replace
    editor, 26
runtime error, 61

save, 47, 52
script, 8
    file, 19
    list, 13
script state, 14
script type, 14
search
    editor, 26
semicolon, 32
send a file, 19
Send As, 15, 19
serialization, 50
SIS file, 22
SMS, 36
sms.delete, 38
sms.get, 37
sms.receive, 37
sms.send, 38
Standalone application, 22

subfolder, 14

trim, 38

ui.cmd, 45
ui.error, 42
ui.form, 41
ui.list, 40
ui.menu, 44
use, 37

variable, 32
view size, 15, 18, 25

website, 5
while, 34
Whole w., 26
Windows, 68
WriteAll, user permission, 30
WriteApp, user permission, 30
WriteDoc, user permission, 30

ZIP files, 20