Programming Tools

Version 3.05

m Mobile Shell, Programming Tools, Version 3.05
Written by Lukas Knecht

www.m-shell.net

Document AB-M-TOO-869

© 2004-2010 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

© 2010 airbit AG

1. Introduction

This manual describes the programming tools available in **m**. The tools are a collection of **m** modules which aid in developing **m** applications and module libraries.

The following tools are currently available:

- Debugging Support (module `debug` (p. 5)) allows to load, compile and execute an **m** process from within another **m** process, set breakpoints, execute it stepwise, examine its variables, etc.
- Generating Documentation (module `mdoc` (p. 20)) produces HTML documentation from a set of **m** source files, similar to `javadoc` or `doxygen`.
- Scanning **m** Sources (module `mscanner` (p. 21)) tokenizes an **m** source, applying the same rules as the scanner used by **m** itself.

2. Debugging Support

The **m** debugging support consist of three parts:

1. The compiler supporting a special debug mode making code and data references available to other processes.
2. The **m** virtual machine offering special execution modes for single stepping and breakpoint catching.
3. The module `debug` serving as the developer's low-level interface to the above.

The **m** installation currently does not contain an actual debugger application. Creating one based on module `debug` should however be straightforward.

Please note that the debugging facilities are only available in the full **m** installation. The **m** environment does not contain the module `debug`, nor the compiler required to load a script in debug mode.

Compiled scripts (`.mex` files) cannot be debugged, only scripts for which the sources are available.

2.1 Module `debug`: Runtime Debugging

This module allows to create a debuggable process and provides access to its data and code.

The typical sequence of calls is:

1. Open a script for debugging with `debug.open` (p. 13), producing a *debuggable process*.
2. Compile and load the code of the process via `debug.compile` (p. 12).

3. If compilation was successful, information about the code units (modules, classes, functions) can be obtained via `debug.ccount` (p. 11), `debug.cindex`, `debug.cinfo`.
4. Likewise, information about the variables (module variables, class member variables, function parameters and local variables) can be obtained via `debug.vcount` (p. 14), `debug.vindex`, `debug.vinfo`.
5. Set or clear breakpoints as desired with `debug.breakp` (p. 10).
6. Execute the process with `debug.go` (p. 13), either stepwise, or until a breakpoint is hit, or until it is paused asynchronously with `debug.pause` (p. 13).
7. Check the process state, and check for runtime errors with `debug.state` (p. 14).
8. Examine the current position in code with `debug.fcount` (p. 12) and `debug.pos` (p. 13).
9. Examine the variable values and types with `debug.vglobal` (p. 15) and `debug.vlocal` (p. 16).
10. Close the process via `debug.close` (p. 10).

2.1.1 Module, Class, Function and Variable Indices

All information about the debuggable process is accessed via indices:

- A module index `mod` denotes a module. `mod=0` is always the main module (script module).
- A class index `cls` denotes a class within a module. `cls=0` refers to the functions and variables of the module itself.
- A function index `func` denotes a function of a module or a class:

`cls`	Denoted function
`0`	Function in module `mod`.
`>0`	Function in class `cls` of module `mod`.

- A variable index var denotes a variable of a module, class or function:

cls	func	Denoted variable
0	<0	Global variable of module mod.
0	>=0	Local variable or parameter of function func of module mod.
>0	<0	Member variable of class cls of module mod.
>0	>=0	Local variable or parameter of function func of class cls of module mod.

A negative index stands for ''Unknown'' or ''not applicable''. See also the function descriptions below.

All functions of module debug throw ExcIndexOutOfRange if the item with a given index does not exist.

2.1.2 Code Positions and Frames

Code positions always refer to a character position in the corresponding source file (which may be different from the character's byte position). New lines always count as one character, regardless of their actual representation in the file.

In short, a *code position* rather corresponds to a document position in a text editor or viewer than to a file position.

When execution of the debuggable process is paused, the current code position is within a series of nested function calls. Each function call corresponds to a *frame*. frame=0 always refers to the topmost frame, i.e. the currently executing function. frame=1 refers to the caller of the currently executing function, and so on. The number of frames can be obtained with debug.fcount (p. 12). debug.fcount(process)-1 therefore always refers to the frame of the main code.

2.1.3 Example

Assume the following (not particulary useful) script SampleScript:

```
use ui
const DEFAULTMUL=20
class C
  p
  function init(p=..DEFAULTMUL)
    this.p=p
  end
  function mul(n)
    return n*p
  end
end
function fact(n)
  if n>1 then
    res=n*fact(n-1)
  else
    res=1
  end;
  return res
end
c:C=C(13)
n=ui.query("n","Debuggable",10)
print "n=" + n
print "fact=" + c.mul(fact(n))
print "fact2=" + c.mul("a")
```

Load this script with the runtime debug module:

```
// open and compile the script
d=debug.open("SampleScript")
debug.compile(d)
// list information about the modules
for i=0 to debug.ccount(d)-1 do
  print debug.cinfo(d,i)
end
→ [C:\documents\mShell\SampleScript.m,-1,0,-1,-1,,1]
   [,-1,1,-1,-1,ui,1]
// get information about fact() (module 0, "class" 0)
fact=debug.cinfo(d,0,0,debug.cindex(d,"fact",0,0));
print fact
→ [C:\documents\mShell\SampleScript.m,155,0,0,40,fact,0]
```

Now run it, demonstrating breakpoints and single stepping:

```
// set a breakpoint at the start of fact()
debug.breakp(d,fact["mod"],fact["pos"])
// run the script, hitting the breakpoint three times
for i=1 to 3 do
  debug.go(d); debug.wait(d)
end
// list the call frames and the values of n in fact()
n=debug.vindex
  (d,"n",fact["mod"],fact["cls"],fact["func"]);
for i=0 to debug.fcount(d)-1 do
  print debug.pos(d,i);
  if debug.pos(d,i)["func"]=fact["func"] then
    print "n=",debug.vlocal(d,n,i)
  end
end
→ [C:\documents\mShell\SampleScript.m,155,0,0,40]
   n= [8,Number]
   [C:\documents\mShell\SampleScript.m,168,0,0,40]
   n= [9,Number]
   [C:\documents\mShell\SampleScript.m,168,0,0,40]
   n= [10,Number]
   [C:\documents\mShell\SampleScript.m,282,0,0,0]
// single step from here
debug.go(d,debug.step); debug.wait(d);
print debug.pos(d)
[C:\documents\mShell\SampleScript.m,168,0,0,40]
// get the index of global variable c
c=debug.vindex(d,"c",0);
// print its value and type
print debug.vglobal(d,0,c)
→ [.C(p=13),Instance,0,1]
// print the value and type of c.p
C=debug.cindex(d,"C",0);
p=debug.vindex(d,"p",0,C);
print debug.vglobal(d,0,c,[p])
→ [13,Number]
```

Demonstrate how exceptions can be handled (''post-mortem debugging''):

```
// remove the breakpoint and run to termination
debug.breakp(d,fact["mod"],fact["pos"],false);
debug.go(d); debug.wait(d)
// find that an exception was thrown
print debug.state(d)
→ ExcStringNotNumber: Operand is a string,not a number
// list the call stack
for i=0 to debug.fcount(d)-1 do
  print debug.pos(d,i)
end
→ [C:\documents\mShell\SampleScript.m,119,0,2,1]
   [C:\documents\mShell\SampleScript.m,313,0,0,0]
// variables are still accessible, so
// get the value of n in C.mul
mul=debug.cindex(d,"mul",0,C);
n=debug.vindex(d,"n",0,C,mul);
print debug.vlocal(d,n)
→ [a,String]
```

debug.breakp

- `function breakp(process,mod,pos,enabled=true)→ Number`

Enables (if `enabled=true`) or disables (if `enabled=false`) the breakpoint in module `mod` at code position `pos` in the debuggable `process`. Execution will pause if an enabled breakpoint is reached.

Returns the real position where the breakpoint was enabled or disabled (breakpoints can only be set at the beginning of statements).

Throws `ErrNotFound` if the code position does not exist. Throws `ErrInUse` if `process` is executing.

debug.close

- `function close(process)→ null`

Closes the debuggable `process`. Its state changes to `debug.died`.

debug.ccount

- function ccount(process,mod=-1,cls=-1)→ Number

Returns the number of code units (or the maximum index + 1):

mod	cls	Count returned
<0		Number of modules.
>=0	<0	Number of classes in module mod.
>=0	0	Number of functions in module mod.
>=0	>0	Number of functions in class cls in module mod.

debug.cindex

- function cindex(process,name,mod=-1,cls=-1)→ Number

Returns the index of a code unit with name name:

mod	cls	Index returned
<0		Index of module.
>=0	<0	Index of class in module mod.
>=0	0	Index of function in module mod.
>=0	>0	Index of function in class cls in module mod.

debug.cinfo

- function cinfo(process,mod,cls=-1,func=-1)→ Array

Returns information about a code unit:

cls	func	Information returned
<0		About module mod.
>0	<0	About class cls in module mod.
0	>=0	About function func in module mod.
>0	>=0	About function func in class cls in module mod.

The information is returned as an array with the following fields:

Key	Meaning	Type
`file`	Source file path	String
`pos`	Position in source file	Number
`mod`	Module index	Number
`cls`	Class index	Number
`func`	Function index	Number
`name`	Name (e.g. function name)	String
`flags`	Flag bits (see below)	Number
`superMod`	Module index of base class (only if class)	Number
`superCls`	Class index of base class (only if class)	Number

`flags` is a combination of the following values:

- `const` **`isnative`** `= 1` A function or variable defined in a native module.
- `const` **`isinherited`** `= 2` An inherited function (not overwritten).
- `const` **`isconst`** `= 4` A `const` variable.

debug.compile

- `function compile(process)→ null|Array`

Compile the debuggable `process`. Returns `null` if successful, an array with the following fields if an error occured:

Key	Meaning	Type
`msg`	Error message	String
`file`	Source file path with error	String
`pos`	Position in source file	Number

Throws `ErrInUse` if `process` is not stopped.

debug.fcount

- `function fcount(process)→ Number`

Return the number of frames (nested function calls) of the debuggable `process`.

Throws `ErrNotReady` if `process` has not been started. Throws `ErrInUse` if `process` is executing.

debug.go

- function go(process,mode=debug.run,arg="")→ null

Continues execution the debuggable process (or starts it), either until a breakpoint is hit, or the condition implied by mode occurs. If the process is started, arg is passed to it as a parameter.

mode determines the condition when execution pauses:

- const **run** = 0 Execute until a breakpoint is hit, an exception occurs, or the process terminates normally.
- const **stepinto** = 1 Execute the next statement, stepping into function calls.
- const **step** = 2 Execute the next statement, behaving like debug.run within functions called from this frame.
- const **stepout** = 3 Execute until the current function returns, otherwise behaving like debug.run.

Throws ErrInUse if process is not compiled or paused.

debug.open

- function open(name)→ Native Object

Opens the script with name name to debug it, creates a new debuggable process, and returns it. The script is not yet compiled or loaded.

Throws ErrNotFound if the script does not exist.

debug.pause

- function pause(process)→ null

Pauses the debuggable process, as if a breakpoint has been hit.

debug.pos

- function pos(process,frame=0)→ Array

Get the current position of execution for the frame with index frame. The position is returned as an array with the following fields:

Key	Meaning	Type
`file`	Source file path	String
`pos`	Position in source file	Number
`mod`	Module index	Number
`cls`	Class index	Number
`func`	Function index	Number

Throws `ErrNotReady` if `process` has not been started. Throws `ErrInUse` if `process` is executing.

debug.state

- `function state(process)→ Number|String`

Get the state the debuggable `process` is in.

If a string is returned, execution terminated with an exception, and the return value is the expression thrown (normally the error message).

If a number is returned, the debuggable is in one of the following states:

- `const` **`opened`** `= 0` The process has been opened with `debug.open`.
- `const` **`compiled`** `= 1` The process has been compiled successfully with `debug.compile`.
- `const` **`running`** `= 2` The process is running and executing code.
- `const` **`waiting`** `= 3` The process is running and waiting from some asynchronous external event (input, timeout, message arriving).
- `const` **`paused`** `= 4` The process is not running (a breakpoint was hit, or an execution step was completed). It can be continued with `debug.go` (p. 13).
- `const` **`ended`** `= 5` The process terminated, either normally or with an exception. Variables can still be examined.
- `const` **`died`** `= 6` The process has been closed and was removed.

debug.vcount

- `function vcount(process,mod,cls=0,func=-1)→ Number`

Returns the number of variables (or the maximum variable index + 1):

cls	func	Count returned
0	<0	Number of global variables in module mod.
0	>=0	Number of local variables and parameters in function func in module mod.
>0	<0	Number of member variables in class cls in module mod.
>0	>=0	Number of local variables and parameters in function func in class cls in module mod.

debug.vglobal

- function vglobal(process,mod,var,follow=[])→ Array

Returns value and type of the global variable var in module mod of the debuggable process. If the value is an array or a class instance, its elements or member variables are derefenced in order by the element indices or member variable indices in follow.

The array returned has the following elements:

Index	Meaning	Type
0	Value of variable	Number or String
1	Type of variable	String
2	If Array, the number of elements	Number
2	If Instance, the module index of its class	Number
3	If Instance, the class index of its class	Number

Note that non-numeric values are always returned as strings, since data from the debugged process cannot be accessed directly from the debugging process.

The possible types are Number, String, Array, Native, Boolean, null, Instance, Function, InstanceFunction.

Throws ErrNotReady if process has not been started. Throws ErrInUse if process is executing.

debug.vindex

- function vindex(process,name,mod,cls=0,func=-1)→ Number

Returns the index of a variable with name name:

cls	func	Index returned
0	<0	Index of global variable in module mod.
0	>=0	Index of local variable or parameter in function func in module mod.
>0	<0	Index of member variable in class cls in module mod.
>0	>=0	Index of local variable or parameter in function func in class cls in module mod.

debug.vinfo

- function vinfo(process,mod,cls,func,var)→ Array

Returns information about a variable with index var:

cls	func	Information returned
0	<0	About global variable in module mod.
0	>=0	About local variable or parameter in function func in module mod.
>0	<0	About member variable in class cls in module mod.
>0	>=0	About local variable or parameter in function func in class cls in module mod.

The information is returned as an array with the same fields as those returned by debug.cinfo (p. 11):

Key	Meaning	Type
file	Source file path	String
pos	-1	Number
mod	Module index	Number
cls	Class index	Number
func	Function index	Number
name	Name of variable	String
flags	Flag bits	Number

debug.vlocal

- function vlocal(process,var,frame=0,follow=[])→ Array

Returns value and type of the local variable with index var in the function at frame frame of debuggable process. If the value is an array or a class instance, its elements or member variables are derefenced in order

by the element indices or member variable indices in follow.

See debug.vglobal (p. 15) for information about the array returned.

Throws ErrNotReady if process has not been started. Throws ErrInUse if process is executing.

debug.wait

- function wait(process,timeout=-1)→ Number|String

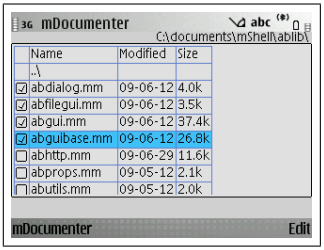
Waits until the debuggable process pauses or terminates, and returns its new state, like debug.state (p. 14).

If timeout>=0 and timeout milliseconds have passed without the script pausing, the current state is returned.

3. Generating Documentation

Generating documentation from **m** source is a simple way of producing reference documentation, in particular for APIs. If a source is commented properly and marked up where necessary, helpful HTML documentation can be generated automatically.

- The module `mdoc` scans list of **m** sources (and the sources imported by them) and generates HTML documentation from them. The generated documentation is configurable by class properties, and of course by subclassing and overriding the HTML generating methods.
- A sample application `mDocumenter` adds a user interface on top of module `mdoc` to select source files and generate documentation. All the documentations of **m** modules in the mdoc index have been generated using this tool.



Sample **m** *screen*

3.1 Item Descriptions

Each item (module, class, function, variable) may be described in the source by a comment. A comment between `/*` and `*/` *before* an item

is considered the item's description and included into the collected data and generated output. For the module or script itself, it is the comment the module source starts with.

The comment is output ''as is'', i.e. can be marked up with HTML. The following `@`-tags in comments are specially processed:

- `@internal`: if the only word in the comment, output of the entire item is suppressed.
- `{@link item}`: insert a link to the item. See the corresponding mdoc.
- `@param p`: explains the function parameter with name `p`.
- `@return`: explains the return value of a function.

Here is an example:

```
/*
  Run the event handling loop. This updates the UI,
  handles UI commands, idle timer ticks, and, if set,
  reads from streams. Returns if either {@link exit()}
  has been called, or a menu command has been executed.
  @return the menu command, or null if {@link exit()}
  has been called.
*/
function run()
  ...
```

3.2 Module `mdoc`: Documentation Generator

This module is part of the `mDocumenter` package which can be downloaded from www.m-shell.net/download.aspx.

Please refer to the mdoc of module `mdoc` for reference information.

4. Scanning m Sources

Scanning an **m** source means reading the tokens (keywords, identifiers, separators, string and number literals, comments) one after another. Scanning is the first step when analyzing **m** sources, e.g. extracting static metrics, extracting comments, instrumenting it, etc.

The module `mscanner` is used by module `mdoc` (p. 20) to read an **m** source and all the files it imports.

4.1 Module `mscanner`: Source Scanner

This module is part of the `ablib msource` package which can be downloaded from www.m-shell.net/download.aspx.

Please refer to the mdoc of module `mscanner` for reference information.

Index