

Native Module Interface

Version 3.07



m Mobile Shell, Native Module Interface, Version 3.07
Written by Lukas Knecht

www.m-shell.net

Document AB-M-NMI-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**

2 DLL Structure **5**
2.1 World Server Example 5
2.2 Module Class 6
2.2.1 Module Versioning 8
2.2.2 Defining Module Members 9
2.2.3 Implementing Functions 11
2.3 DLL Entry Points 12

3 m Runtime System **15**
3.1 **m** Data Types and Values 15
3.1.1 Scalars 15
3.1.2 Strings 17
3.1.3 Arrays 18

4 Creating, Installing and Testing the module DLL **23**
4.1 Compiling and Linking 23
4.2 Installing 25
4.2.1 Symbian 2nd Edition 25
4.2.2 Symbian 3rd/5th Edition 26
4.3 Testing 26

5 Advanced Concepts **27**
5.1 Alarm Server Example 27
5.2 User Permissions 30

5.3 Asynchronous Request Handling 31
5.3.1 Timeouts 34
5.4 Native Objects 35
5.5 Native Streams 37
5.6 Native Icons 39
5.7 Memory Allocation 39
5.8 Module Loading and Initialization 40
5.9 Restrictions 41

A NMI SDK **43**

B Platform Security **47**

Index **49**

1. Introduction

The **m** Native Module Interface (NMI) specifies how **m** interacts with modules written entirely in C++ for Symbian OS.

Each native module corresponds to a (''polymorphic'') dynamic link library (DLL) in Symbian OS. This document describes the conventions imposed by the **m** native module interface, and how native code interacts with **m**'s runtime system.

To write native modules, the NMI Software Development Kit (SDK) must be installed. See appendix A (p. 43). Note that the SDK for Symbian only contains an ARMI runtime library. Linking and testing in the emulator is not supported.

This manual is organized as a tutorial, explaining each step of writing, compiling and installing a native module.

Throughout this manual, two native modules are developed:

1. A module accessing the Symbian ''world server'', a database of cities and countries built into Symbian devices. This example shows the basic concepts of a native module: how functions are declared and implemented, and how **m** data is accessed and created.

2. A module accessing the Symbian ''alarm server'', with elementary functions to manipulate alarms and to wait for the next due alarm. This example shows how to deal with asynchronous functions and how to add a timeout to such a function.

The complete source for the two modules developed is also part of the SDK.

Most of the information applies to Symbian 2nd Edition and Symbian 3rd/5th Edition. Where there are differences, they are marked accordingly.

2. DLL Structure

In the simplest (and most typical) case, an NMI DLL consists of a single C++ source defining a single class and two functions serving as DLL entry points.

2.1 World Server Example

The concepts of this and the next two chapters are illustrated by developing a module accessing the Symbian ''world server''. The client API to this server is the Symbian class `RWorldServer`. `WorldModule.cpp` contains the complete source of the module and is part of the NMI SDK.

We call our module `world`[1], and after studying the services offered by the world server API, we specify it as follows:

1. The module `world` should provide the following functions:
   - `world.count()→Number` returns the number of cities in the database.
   - `world.find(city,country=null)→Array` searches for the first city with the given name or name prefix. The search can optionally be limited to a country.
   - `world.first()→Array` returns the first city in the database.
   - `world.home()→Array` returns the configured home city of the phone owner.
   - `world.next()→Array` returns the next city in the database.
2. A city should be returned as an array with the following members:

[1] A generally useful module (e.g. one exhibiting an OS API) should have a short, reasonably clear, but potentially unique name. Note that all module names share the same name space, so be careful when naming your module to avoid potential conflicts.

Key	Contents
`name`	The city's name.
`nat`	The city's nationial phone number code.
`area`	The city's area code.
`utcofs`	The city's time zone as an offset to UTC.
`lat`	The latitude of the city's position.
`long`	The longitude of the city's position.
`country`	The name of the country the city is in.
`capital`	The name of the country's capital.

If the city does not exist, the functions should return `null`.

3. For the sake of this example, we would also like the module to maintain a variable `world.calls` counting the number of calls we make to the module.

2.2 Module Class

Each native module is implemented as a class inheriting from abstract class `NativeModule`, defined in `NativeModule.h`. It **must** implement the following abstract virtual functions:

- `virtual TInt ExpectedRuntimeVersion();`
  This method is used by the NMI framework to check for the correct runtime system version. It should simply return `Runtime::VERSION` (a compile time constant).

- `virtual void ConstructL();`
  This is the 2nd phase constructor for the native module. It will typically perform module specific initialization which should happen when the module is loaded, then define the functions, variables and constants provided by the module.

- `virtual Runtime::Value ExecuteL(TInt index, Runtime::Value *params, TInt paramCount, TRequestStatus &status);`
  This is the function which actually performs the native function. `index` identifies the desired function, `params` and `paramCount` are the parameters passed, and `status` is the asynchronous request

status to be used for asynchronous calls. The function returns the result of the **m** function (unless an asynchronous call was made, see section 5.3 (p. 31)).

If `ExecuteL()` leaves (e.g. via `User::Leave`), the **m** runtime system catches the leave and converts it into an **m** runtime exception.

In addition, it **can** implement the following virtual functions:

- `virtual const char *ModuleVersion();`
  Get the version of this module. This should return a number in major.minor notation. Surrounding non-numeric characters are trimmed. If it returns NULL (the default), the module version is set to 0.0.

- `virtual void Cancel();`
  Cancel any outstanding asynchronous request started in `ExecuteL()`. The default implementation does nothing.

- `virtual TInt HandleError(TInt error);`
  Handle an error of an asynchronous request started in `ExecuteL()`. This should return `KErrNone` if the error will be handled by the next call to `ExecuteL()`, otherwise return an error code to be handled by the runtime system. The default implementation simply returns `error`.

- `virtual const TDesC& GetErrorMessage(TInt error);`
  Get a human readable translation for an error code produced by functions in this module, or an empty string for a default message. The default implementation returns the empty string.

- `virtual TBool HandleProcessEventL(TEventCode type, const TAny *param);`
  Handle an event on the process console or UI. This can be used to show and hide alternative views on top of the process view, or to otherwise react to console being hidden or shown. The default implementation does nothing. See also `Runtime::ProcessEvent`.

The only variable visible to subclasses is:

- `protected: Runtime *runtime;`
  A pointer to the **m** runtime system, providing functions to create and manipulate **m** objects and to access the application.

World Server Example

We start our ''world'' module by including the necessary files and defining the skeleton of our class. Our module will maintain a connection to the world server, i.e. an instance of the Symbian class `RWorldServer`, defined in `t32wld.h`. Note that all class members can be kept private, except for the implementation of the virtual functions defined in `NativeModule`:

```
#include "NativeModule.h"
#include <t32wld.h>

class WorldModule : public NativeModule {
private:
  // @todo: some declarations

  // a connection to the Symbian world server, and the current city id
  RWorldServer server;
  TWorldId id;

  // destroying the module should close the server
  ~WorldModule() {
    server.Close();
  }

protected: // from NativeModule
  TInt ExpectedRuntimeVersion() { /* @todo */ }

  const char *ModuleVersion() { /* @todo */ }

  void ConstructL() { /* @todo */ }

  Runtime::Value ExecuteL(TInt index, Runtime::Value *params,
                          TInt paramCount, TRequestStatus &status)
  { /* @todo */ }
};
```

We will now fill the `@todo` sections step by step.

2.2.1 Module Versioning

Since neither C++ nor Symbian offer a proper interface or DLL versioning mechanism interface, we must somehow ensure that a DLL we load really is an **m** module. This is achieved by implementing the following

function defined in `NativeModule`:

```
TInt ExpectedRuntimeVersion() { return Runtime::VERSION; }
```

`Runtime::VERSION` is guaranteed to change if the interface defined by the `Runtime` class changes. Modules relying on another version of the runtime system can therefore be identified and prevented from initializing before harming the **m** application.

Furthermore, there is a simple module versioning mechanism built into **m**: Each module has its `..version` constant. For a native module, this version is the string returned by `ModuleVersion()`. If you manage your versions via CVS, you can automatically have CVS insert the current revision, for instance

```
const char *ModuleVersion() { return "$Revision: 887 $"; }
```

The **m** runtime system will trim non-numeric characters and just assign `887` to `..version`.

2.2.2 Defining Module Members

As outlined above, each function is identified by its index (a unique integer). The simplest way to define those indices is to define an (anonymous) enumeration. The functions and any global module variables and constants are then added to m's internal tables in the 2nd phase constructor of the class, in its `ConstructL()` function, using the following functions from `class Runtime`:

- `void AddNativeFunctionL(const TDesC &name, TInt minParamCount, TInt maxParamCount, TInt index);`
  Adds a function by linking it to its index, and specifies the minimal and maximal number of arguments permitted. The m compiler can thus perform some (minimal) checks when compiling a call to the function.

- `Runtime::Value &GetVariableL(const TDesC &module, const TDesC &variable, TBool isConst = EFalse);`
  Adds a global variable to a module and returns a reference to it. If the variable does not exist, it is created and initialized to `null`.

- `Runtime::Value &AddConstantL(const TDesC &name, const TDesC &value);`
  `Runtime::Value &AddConstantL(const TDesC &name, TReal value);`
  `void AddConstantL(const TDesC &name, Runtime::Value value);`
  Add a constant to the module, e.g.

```
runtime->AddConstantL(_L("answer"), 42);
```

World Server Example, Continued

For the functions and the `calls` variable of our ''world'' module, defining module members looks as follows:

```
// the functions this module supports
// the enumeration values correspond to function indices
enum {
  CountFunction, FindFunction, FirstFunction, HomeFunction, NextFunction
};
```

Since we use them more than once, we also declare the module and variable name descriptor constants for the `world.calls` variable:

```
// the names to access the calls variable
_LIT(WORLD, "world");
_LIT(CALLS, "calls");
```

```
void ConstructL() {
  // the five functions this module supports
  runtime->AddNativeFunctionL(_L("count"), 0, 0, CountFunction);
  runtime->AddNativeFunctionL(_L("find"), 1, 2, FindFunction);
  runtime->AddNativeFunctionL(_L("first"), 0, 0, FirstFunction);
  runtime->AddNativeFunctionL(_L("home"), 0, 0, HomeFunction);
  runtime->AddNativeFunctionL(_L("next"), 0, 0, NextFunction);
  // initialize the variable counting the function calls
  runtime->GetVariableL(WORLD, CALLS).SetNumber(0);
}
```

None of our functions take arguments except `world.find`, which can have one or two.

The value reference returned by `runtime->GetVariableL()` is set to (the number) zero (the C++ type of an **m** variables is `Runtime::Value`, which will be explained in detail in section 3.1 (p. 15)).

2.2.3 Implementing Functions

Each native module has only a single entry point for all the functions it declares: the virtual function `ExecuteL()`. The actual function called is passed as a parameter, and corresponds to the index set when adding the function via `runtime->AddNativeFunctionL()`.

Besides the function index, `ExecuteL()` gets the following parameters passed by the **m** runtime system:

- `params`: a pointer to the parameter array, `params[0]` corresponds to the first parameter, `params[1]` to the second, an so on.
- `paramCount`: the number of parameters passed in the actual call. This is guaranteed to be in the bounds specified by the corresponding `runtime->AddNativeFunctionL()` call.
- `status`: a reference to the asynchronous request state. Only functions performing asynchronous requests will have to deal with this parameters; this is explained in section 5.3 (p. 31).

`ExecuteL()` returns the `Runtime::Value` which is the result of the **m** function.

World Server Example, Continued

The skeleton for the functions of our module `world` might therefore look as follows:

```
Runtime::Value ExecuteL(TInt index, Runtime::Value *params,
                        TInt paramCount, TRequestStatus &status)
{
  // if the world server is not connected, connect it
  if (server.Handle() == 0) User::LeaveIfError(server.Connect());
  // a variable which will contain the function result
  Runtime::Value result;
  // @todo: increment calls variable
  switch (index) {
  case CountFunction:
    // @todo
  case FindFunction:
    // @todo
  case FirstFunction:
    // @todo
  case HomeFunction:
    // @todo
  case NextFunction:
    // @todo
  }
  return result;
}
```

Note that the server connection is not established until the first call of `ExecuteL()`, i.e. until a function of the module is called. The reasoning behind this is explained in section 5.8 (p. 40).

The next chapter will replace the `@todo` comments by actual code.

2.3 DLL Entry Points

Each native module DLL must implement and export functions which serve as entry points into the DLL: the first will be called by the **m** runtime system to create the module instance, the second is only needed in Symbian 2nd Edition, imposed by the rules for DLLs. Symbian 3rd/5th Edition defines the symbol `EKA2` (EPOC Kernel Architecture 2) which can be used to omit the function.

World Server Example, Continued

In our `world` module, the two entry point functions read as follows:

```
// the first exported function creates an instance of this module
EXPORT_C NativeModule* NewWorldModuleL() {
  return new(ELeave) WorldModule;
}

// the second exported function is required by DLL conventions
// (Symbian 2nd Edition only)
#ifndef EKA2
GLDEF_C TInt E32Dll(TDllReason) { return KErrNone; }
#endif
```

For a DLL, the name of the first exported function (`NewWorldModuleL` here) does not matter. However, to support static linking in standalone applications[2], the function name must be of the form `New`*`Name`*`ModuleL`, where *`Name`* ist the name of your module, with the first character uppercase and all others lowercase. See chapter 4 (Reference, p. 61) for more information about producing statically linked standalone applications.

[2]Static linking is currently restricted to the standard modules of **m**. If you want to add your own native modules to a statically linked **m** application, you must add them as DLL.

3. m Runtime System

The **m** runtime system defines the interface for a native module to its **m** process, to access variables and values and to create new ones. It consists of the following classes:

- `class Runtime`: the interface to the runtime system, consisting of some 50 functions, including the ones adding functions, variables and constants which have already been presented.
- `union Runtime::Value`: a value (number, string, array, etc.) in **m**.
- `struct Runtime::Array`: an (associative) **m** array.
- `struct Runtime::ValueArray`: a (non-associative) flat, dynamic array of `Runtime::Value`.
- `struct Runtime::String`: an (immutable) **m** UNICODE® string.
- `struct Runtime::NativeObject`: a reference to a native (C++ or Symbian) object which is deleted by garbage collection.

They are all declared in file `Runtime.h`. Consider `Runtime.h` as the reference when writing native **m** modules.

3.1 m Data Types and Values

3.1.1 Scalars

m is not a strongly typed language. As such, it allows variables and array elements to assume any valid **m** data type (number, string, array, etc.) and to change that type at any time (positively put, **m** is ''fully polymorphic''). Consequently, all variables and array elements in **m** are

of type `Runtime::Value`. Internally, this is a union of eight bytes aligned on an eight byte boundary.

The following table summarizes the methods to set and get a `Runtime::Value`:

m type	C++ type	Getter	Setter/Creator
`Number`	`TReal`	`GetNumberL()`	`SetNumber()`
`Number`	`TInt`	`GetIntL()`	`SetNumber()`
`Number`	`TTime`	`GetTimeL()`	`SetTime()`
`Number`	`TTimeIntervalMicroSeconds32`	`GetTimeoutL()`	`SetTimeout()`
`Number`	`TUid`	`GetUidL()`	
`Boolean`	`TBool`	`GetBooleanL()`	`SetBoolean()`
`String`	`Runtime::String`	`GetStringL()`	`NewStringL()`
`String`	`TPtrC`	`GetPtrCL()`	
`Array`	`Runtime::Array`	`GetArrayL()`	`NewArrayL()`
`null`		`IsNull()`	`SetNull()`

For most datatypes, there are also constructors or equivalent static methods, e.g. `Runtime::BooleanValue`.

The getters all leave if the type of the value is not the requested one. You can use `IsNumber()`, `IsString()` etc. to check for the type of a value.

`GetTimeoutL()` and `SetTimeout()` convert a millisecond number to a microsecond timeout value and vice versa. `GetTimeoutL()` also checks for the number not exceeding $2^{31} - 1$ microseconds (2147483 milliseconds).

`GetUidL()` correctly handles the signed/unsigned problems caused by Symbian's inconsistent definition of the `TUid` type: numbers in the ranges `0x80000000` to `0xffffffff` and `-0x7fffffff` to `-1` are both mapped to the correct `TUid` instance.

World Server Example, Continued

Returning to our world server example, we can already count the function calls and implement `world.count` with the above table:

```
// count this call
Runtime::Value &calls = runtime->GetVariableL(WORLD, CALLS);
calls.SetNumber(calls.GetNumberL() + 1);
switch (index) {
case CountFunction:
  // directly set the result
  result.SetNumber(server.NumberCities());
  break;
```

Furthermore, even though we don't know yet how to create the **m** array containing a city's data, we can declare the following function which we will need to create the **m** result of our functions:

```
/*
  Return the current city (defined by this->id) as an associative m
  array.
  @param error the outcome of the world server call: KErrNone if the
  city exists, KErrNotFound if the city does not exist; any other code
  indicates an error.
  @return the m array with the city's data.
*/
Runtime::Value GetCurrentCityL(TInt error) {
  // @todo
}
```

3.1.2 Strings

Strings in **m** are immutable arrays of 16-bit characters, just as in Symbian. They are created via one of the `runtime->NewStringL()` overloads. For instance, the following fragment creates four times the string `"NMI"`:

```
Runtime::Value a = runtime->NewStringL("NMI");
Runtime::Value b = runtime->NewStringL(_L("NMI"));
Runtime::Value c = runtime->NewStringL(_L8("NMI"));
Runtime::Value d = runtime->NewStringL(3);
TText* s = d.GetStringL().GetBuf();
s++ = 'N'; s++ = 'M'; s++ = 'I';
```

To read a string, use `Runtime::Value::GetPtrCL()`, which returns a Symbian `TPtrC`, and leaves if the value is not a string:

```
Runtime::Value a = runtime->NewStringL("NMI");
TBuf<32> buf;
buf.Format(_L("a=%S"), &a.GetPtrCL()); // buf = "a=NMI"
```

See the `struct Runtime::String` definition in `Runtime.h` for additional functions on **m** string values.

World Server Example, Continued

We can now implement all functions of our world server module:

```
case FindFunction:
  {
    // get the first argument, leaving if it is not a string
    TPtrC city(params[0].GetPtrCL());
    result = GetCurrentCityL
      (paramCount == 2 && !params[1].IsNull()
       // with country (2nd parameter which is not null)
       ? server.Find(id, city, params[1].GetPtrCL())
       // without country (no 2nd parameter, or it is null)
       : server.FindCity(id, city));
  }
  break;
case FirstFunction:
  result = GetCurrentCityL(server.FirstCity(id));
  break;
case HomeFunction:
  result = GetCurrentCityL(server.Home(id));
  break;
case NextFunction:
  result = GetCurrentCityL(server.NextCity(id));
  break;
```

In the implementation of `world.find`, `params[0].GetPtrCL()` obtains the first parameter passed to the function as a string, leaving if it is not a string. We then check whether to call `WorldServer::Find` or `WorldServer::FindCity`, based on the presence and the value of the second parameter `params[1]`.

Note that, in conjunction with the parameter count range set via `runtime->AddNativeFunctionL()`, all invalid parameters are catched, causing the function to leave. The **m** runtime system will convert the Symbian leave into an **m** exception.

3.1.3 Arrays

Arrays are a fundamental data structure in **m**: since **m** values are polymorphic, arrays also serve to represent flat or arbitrarily complex and even recursive data structures.

Arrays are created via one of the `runtime->NewArrayL()` overloads. For instance, the following fragment creates the **m** array `["The", "answer", "is", 42]`:

```
Runtime::Value a = runtime->NewArrayL();
Runtime::Array &array = a.GetArrayL();
array.AppendL(runtime->NewStringL("The"));
array.AppendL(runtime->NewStringL("answer"));
array.AppendL(runtime->NewStringL("is"));
array.AppendL(Runtime::Value(42));
```

Since we know there are four elements in the array, it is slightly more efficient to pre-allocate space for exactly four elements. Also, we can directly obtain a pointer to the new array, omitting the `GetArrayL()` call and the unnecessary check it performs:

```
Runtime::Array *array;
Runtime::Value a = runtime->NewArrayL(array, 4);
array->AppendL(runtime->NewStringL("The"));
...
```

To access array elements, there are a number of functions. All functions perform array boundary checking, leaving if the index is out of bounds. The following fragment changes the above array into `["The", "question", "is", "unknown"]`:

```
array->SetL(1, runtime->NewStringL("question"));
array->SetL(3, runtime->NewStringL("unknown"));
```

To get or set values with string keys, use the corresponding overloads:

```
array->SetL(_L("extra"), runtime->NewStringL("so far"));
runtime->WriteL(array->GetL(_L("extra")));
```

`Runtime::Array::SetL` with a string key appends an element if the corresponding key does not exist yet, so the array is now equivalent to `["The", "question", "is", "unknown", "extra":"so far"]`. The second line simply writes `"so far"` to the **m** process standard output.

A `Runtime::Array` internally consists of two data structures: a lazily created hash table maintaining the relationships between string keys and array indices, and the actual flat array of values. The flat array is of type `Runtime::ValueArray` and can be accessed directly via `Runtime::Array::GetValues()`. If you perform a lot of operations only on the flat array, it may be more efficient to operate directly on it.

For instance, the following function computes the scalar product of two arrays:

```
Runtime::Value DotProductL(const Runtime::Value &a,
                           const Runtime::Value &b)
{
  Runtime::ValueArray &aa = a.GetArrayL().GetValues(),
    &ab = b.GetArrayL().GetValues();
  TInt len = aa.Length();
  // check the two arrays have the same length
  if (ab.Length() != len) User::Leave(KErrArgument);
  TReal s = 0;
  for (TInt i = 0; i < len; i++)
    s += aa.NumberL(i) * ab.NumberL(i);
  return Runtime::Value(s);
}
```

The calls to `NumberL()` perform array boundary checking. Since we know the boundaries are not violated, we can directly access the flat array as an array of `Runtime::Value`:

```
  Runtime::Value *va = aa.GetPtr(), *vb = ab.GetPtr();
  TReal s = 0;
  for (TInt i = 0; i < len; i++)
    s += va[i].GetNumberL() * vb[i].GetNumberL();
```

See the `struct Runtime::Array` and `struct Runtime::ValueArray` definitions for additional functions on **m** arrays.

World Server Example, Continued

We can now finish our module by implementing `GetCurrentCityL`. Remember that it, depending on the outcome of a call, should either return an associative array with the elements we specified, or `null`, or leave with an error.

```
Runtime::Value GetCurrentCityL(TInt error) {
  Runtime::Value result;
  if (error == KErrNone) { // city exists
    TCityData city;
    User::LeaveIfError(server.CityData(city, id));
    TCountryData country;
    User::LeaveIfError(server.CountryData(country, id));
    // create an array, pre-allocating for eight elements
    Runtime::Array *array;
    result = runtime->NewArrayL(array, 8);
    // append a string with the city name
    array->SetL(_L("name"), runtime->NewStringL(city.iCity));
    // append a string with the nation code
    array->SetL(_L("nat"), runtime->NewStringL(country.iNatCode));
    // append a string with the area code
    array->SetL(_L("area"), runtime->NewStringL(city.iAreaCode));
    // append a number with the UTC offset
    array->SetL(_L("utcofs"),
                Runtime::Value(city.iUniversalTimeOffset));
    // append a number with the latitude in degrees
    array->SetL(_L("lat"),
                Runtime::Value((TReal)city.iLatLong.MinutesLat() / 60));
    // append a number with the longitude in degrees
    array->SetL(_L("long"),
                Runtime::Value((TReal)city.iLatLong.MinutesLong()/ 60));
    // append a string with the country name
    array->SetL(_L("country"), runtime->NewStringL(country.iCountry));
    // append a string with the capital
    array->SetL(_L("capital"), runtime->NewStringL(country.iCapital));
  }
  else if (error == KErrNotFound) // city does not exist
    result.SetNull();
  else // real error
    User::Leave(error);
  return result;
}
```

4. Creating, Installing and Testing the module DLL

Once a module has been implemented, it must be compiled, linked into a DLL, the DLL installed on the device and tested.

How this happens depends to a large extent on your operating system, the SDK installed and whether you use an IDE like Carbide.c++ or not.

There are also differences between Symbian 2nd Edition and Symbian 3rd Edition due to the introduction of platform security, both with respect to compilation and installation.

The following example assumes you use a standard Symbian SDK on Windows, e.g. downloaded from developer.nokia.com. This is often absolutely sufficient to develop native **m** modules. But of course you can also use fancy IDEs or your Linux based Symbian development environment [1]. As the NMI SDK is reasonably simple, configuring it for your environment should also be straightforward.

4.1 Compiling and Linking

A native module must be compiled into a polymorphic DLL file with name `module_mm.dll`, where *module* is the name by which the module is known to **m**. Such a DLL has `UID2=0x1000008d`. You are free to specify an `UID3`. If you do, the **m** runtime environment UID is an option:

UID	Platform, signing type
`0x1020429a`	Symbian 2nd Edition
`0xa0009885`	Symbian 3rd/5th Edition, self signed and DevCert signed
`0xe7e0cab8`	Symbian 3rd/5th Edition, online signed

[1] In fact, **m** for Symbian is completely developed and maintained under Linux.

Each module must be linked:

- for Symbian 2nd Edition, with the Symbian libraries `edll.lib` and `euser.lib`, and the **m** runtime library `mRuntime.lib`.
- for Symbian 3rd/5th Edition, with the Symbian library `euser.dso`, and the **m** runtime library `mRuntime.dso`. The latter comes in two versions for each device type, depending whether you link for a self signed, DevCert signed, or online signed **m** installation.

The other libraries depend on the Symbian APIs your native module uses.

World Server Example, Continued

Assuming the standard SDK, a minimum of three files are needed:

1. The C++ source `WorldModule.cpp`.
2. The `bld.inf` file naming the project file. A minimal file is:

```
PRJ_MMPFILES
World.mmp
```

3. The `World.mmp` file describing the project. An example for Symbian 2nd Edition:

```
// World native module project for Symbian 2nd
TARGET World_mm.dll
TARGETTYPE dll
UID 0x1000008d 0x1020429a
SOURCEPATH .
SOURCE WorldModule.cpp
USERINCLUDE ..\..\include
SYSTEMINCLUDE \epoc32\include
LIBRARY edll.lib euser.lib mRuntime.lib
LIBRARY worldclient.lib
EXPORTUNFROZEN
```

   The same for Symbian 3rd/5th Edition:

```
// World native module project for Symbian 3rd/5th
TARGET World_mm.dll
TARGETTYPE dll
UID 0x1000008d 0xa0009885
CAPABILITY LocalServices NetworkServices ReadUserData \
UserEnvironment WriteUserData
SOURCEPATH .
SOURCE WorldModule.cpp
USERINCLUDE ..\..\include
SYSTEMINCLUDE \epoc32\include
LIBRARY euser.lib mRuntime.lib
LIBRARY worldclient.lib
EXPORTUNFROZEN
```

The capabilities listed are the minimum set for a self signed **m** installation. See B (p. 47) for further information.

Once all the files are in place, just call

```
bldmake bldfiles
abld build armi urel
```

After these two steps, `Epoc32\release\armi\urel` of your SDK should contain a file `World_mm.dll`. This is the native **m** module!

For Symbian 3rd/5th Edition, you must replace the platform `armi` by `gcce`:

```
abld build gcce urel
```

And the DLL file should be in `Epoc32\release\gcce\urel` of your SDK.

4.2 Installing

4.2.1 Symbian 2nd Edition

On Symbian 2nd Edition, a native module DLL should reside in `system.appdir`. The proper way to install it is via a .sis package, but this requires a UID and is not needed for simple testing.

Especially during testing, it is often simpler to transfer the `.dll` file on the device, e.g. via Bluetooth® or a memory card, and to copy it to the application directory via an explorer application or, even simpler, via **m**. The `dllinstall.m` script searches all devices for `*_mm.dll` files and allows to choose one to be copied.

4.2.2 Symbian 3rd/5th Edition

On Symbian 3rd/5th Edition, all DLLs must reside in `\sys\bin` and can only be installed via a signed SIS package.

There are several ways of creating such a package. Because it does not require an extra UID to be obtained from Symbian, the simplest is to create a package file patching **m**, with package type `SP`, and the **m** UID (`0xa0009885` or `0xe7e0cab8`, depending on the signing type). A sample package file `World-S60_3.pkg` for the `world` module looks as follows:

```
&EN
#"m Shell World module",(0xa0009885),2,0,0,TYPE=SP
%"yourself"
:"yourself"
[0x101F7961],0,0,0,"S60ProductID"
"Epoc32\Release\gcce\urel\World_mm.dll"-"!:\sys\bin\World_mm.dll"
```

Note that the DLL source path (`Epoc32\...`) must be prefixed by your SDK installation.

Calling

```
createsis create World-S60_3.pkg
```

will then create a file `World-S60_3.sis` to be transferred to your phone.

On Symbian 5th Edition, the sample package file is `World-S60_5.pkg`. It is identical except for the platform id.

4.3 Testing

Once `world_mm.dll` has been installed, it is ready to be tested:

```
use world
world.find("Bern")
→ [Bern,41,31,60,46.95,-7.466666667,Switzerland,Bern]
```

Of course, systematic testing requires a lot more than just trying a few calls. Writing a complete test script for a native module in **m** is easy and always worth the effort. It greatly simplifies maintenance and helps in resolving portability issues.

5. Advanced Concepts

After successfully mastering the basics of writing and deploying **m** native modules, you will learn in this chapter some of the more advanced concepts in writing native **m** modules.

5.1 Alarm Server Example

The concepts of this chapter are illustrated by developing a module accessing the Symbian ''alarm server''. The client API to this server is the Symbian class `RASCliSession`. `AlarmModule.cpp` contains the complete source of the module and is also part of the NMI SDK.

We call our module `alarm`. And again, after studying the services offered by the alarm server API, we specify it:

1. Alarms are identified by integers (`TAlarmId` is a `TInt`).
2. The module `alarm` should provide the following functions:
   - `alarm.add(msg,due,sound=null)→Number` adds an alarm with message, due time and optional sound file and returns the id of the new alarm.
   - `alarm.all()→Array` returns the ids of all known alarms.
   - `alarm.delete(id)→null` deletes the alarm with the given id.
   - `alarm.get(id)→null` returns some data about the alarm with the given id, as an array with the following members:

Key	Contents
`msg`	The message text of the alarm.
`due`	The next due time of the alarm.
`sound`	The sound file played when the alarm expires.
`enabled`	Whether the alarm is enabled.

- `alarm.on(id, enabled)→null` enables or disables an alarm.
- `alarm.wait(timeout=-1)→Number` waits up to `timeout` milliseconds for the next alarm to expire and returns its id. If `timeout` is negative, it is ignored.

3. The module should respect the permissions granted by the user: `ReadApp` for all alarm functions, `WriteApp` for the functions modifying alarm data.

From this spec, we can directly write the skeleton of our module:

```
#include "NativeModule.h"
#include <asclisession.h>
#include <asshdalarm.h>

class AlarmModule : public NativeModule {
private:
  enum Functions {
    AddFunction, AllFunction, DeleteFunction, GetFunction, OnFunction,
    WaitFunction
  };

  RASCliSession session;
  TASShdAlarm alarm;

  // destroying the module should close the session
  ~AlarmModule() {
    session.Close();
  }
```

Functions are added in the second phase constructor:

```
void ConstructL() {
  // the functions this module supports
  runtime->AddNativeFunctionL(_L("add"), 2, 3, AddFunction);
  runtime->AddNativeFunctionL(_L("all"), 0, 0, AllFunction);
  runtime->AddNativeFunctionL(_L("delete"), 1, 1, DeleteFunction);
  runtime->AddNativeFunctionL(_L("get"), 1, 1, GetFunction);
  runtime->AddNativeFunctionL(_L("on"), 2, 2, OnFunction);
  runtime->AddNativeFunctionL(_L("wait"), 0, 1, WaitFunction);
}
```

Again, the functions themselves are called via `ExecuteL()`:

```
Runtime::Value ExecuteL(TInt index, Runtime::Value *params,
                        TInt paramCount, TRequestStatus &status)
{
  // if the session is not connected, connect it
  if (session.Handle() == 0) User::LeaveIfError(session.Connect());
  // a variable which will contain the function result
  Runtime::Value result;
  result.SetNull(); // default for functions returning null
  switch (index)
  case AddFunction:
    alarm.Reset();
    // copy with truncation to avoid a panic
    alarm.Message().Copy
      (params[0].GetPtrCL().Left(KMaxAlarmMessageLength));
    alarm.NextDueTime() = params[1].GetTimeL();
    if (paramCount > 2 && !params[2].IsNull())
      alarm.SoundName().Copy
        (params[2].GetPtrCL().Left(KMaxAlarmSoundNameLength));
    User::LeaveIfError(session.AlarmAdd(alarm));
    result.SetNumber(alarm.Id());
    break;
  case AllFunction:
    ...

  return result;
}
```

Two details are worth mentioning: first, `TASShdAlarm::Message()` and `TASShdAlarm::SoundName()` return references to fixed size buffers. To avoid a panic when copying longer strings, we truncate them before copying. Second, `Runtime::Value::GetTimeL()` converts an **m** Number into a `TTime` value.

The implementation of the other functions is straightforward Symbian C++ programming, except for `alarm.wait`, which is asynchronous and presented in section 5.3 (p. 31).

The exported entry points are:

```
EXPORT_C NativeModule* NewAlarmModuleL() {
  return new(ELeave) AlarmModule;
}

#ifndef EKA2
GLDEF_C TInt E32Dll(TDllReason) { return KErrNone; }
#endif
```

5.2 User Permissions

Since **m** allows many harmful or costly operations to happen automatically, the user can grant or deny permission for certain operations, for instance to write some files or to dial the phone.

Of course, from within a native module, you have almost complete control over the device, as you can ignore the permissions denied by the user if you want[1]. However, it is a good idea to conform to **m** standards, particularly if you think of writing your module for a broader audience.

Permissions are checked anywhere within `ExecuteL()` by two functions from `class Runtime`:

- `void CheckPermissionsL(TInt access);`
  Checks whether the user granted all permissions in the bit mask `access` and leaves with `ExcNotPermitted` if any of the permissions has been denied. The permission bits are defined in `UserPermissions.h`, which is always included if you include `NativeModule.h`.

- `const TDesC &RealPathL(const TDesC &path, TInt accessMode);`
  Gets the real absolute path of a relative path or file, and checks for user permissions to access it, leaving if the necessary permissions have been denied. This handles `"."`, `".."`, the current directory and missing drives. The path returned need not be valid. The bit mask `accessMode` determines the permissions checked: it can be `ReadAccess`, `WriteAccess` or both.

Alarm Server Example, Continued

In our alarm module, all functions read application data, so we check for `ReadApp` right at the beginning:

[1] **m** user permissions are unrelated to the Symbian 3rd/5th Edition security model which attempts to address this issue to some extent at the OS level with a combination of capabilities and signed installers.

```
Runtime::Value ExecuteL(TInt index, Runtime::Value *params,
                        TInt paramCount, TRequestStatus &status)
{
  // all functions in this module require permission to read
  // application data, we should check for it
  runtime->CheckPermissionL(ReadAppPermission);
  // if the session is not connected, connect it
  ...
```

Furthermore, `alarm.add`, `alarm.delete` and `alarm.on` may write application data, so these functions should check for `WriteApp`:

```
switch (index) {
case AddFunction:
  runtime->CheckPermissionL(WriteAppPermission);
  ...
case DeleteFunction:
  runtime->CheckPermissionL(WriteAppPermission);
  ...
case OnFunction:
  runtime->CheckPermissionL(WriteAppPermission);
```

5.3 Asynchronous Request Handling

Symbian's standard way of dealing with asynchronous requests uses a single threaded model. Events are serializated via ''active objects'', implemented by the Symbian class `CActive` and supporting classes. The **m** runtime follows this scheme: each **m** process is an active object; the **m** script runs in the process' `CActive::RunL()` function.

The asynchronous request status of the process object is passed to `ExecuteL` as `TRequestStatus &status`, allowing to issue asynchronous requests. But what happens if such a request completes? The answer is simple: `ExecuteL()` is called a again with exactly the same parameters.

Step by step, an asynchronous request is started and processed as follows:

1. `ExecuteL()` is called as usual.
2. A function starts an asynchronous request, passing the `status` reference to the requesting function.
3. To signal to the **m** runtime system that an asynchronous request

was started, `ExecuteL()` must return with an ''uncomplete'' result, which is obtained via `Runtime::Value::SetUncomplete()`.

4. The runtime system deals with the other issues of active objects, calls `SetActive()`, suspends the **m** process, and so on.

5. Once the request completes (with or without error), the **m** runtime system continues the **m** process.

6. If the request completed with any result other than `KErrNone`, the module's `HandleError()` function is called to give it an opportunity to handle the error as a normal situation and return `KErrNone`.

7. If the request result is or was mapped by `HandleError()` to `KErrNone`, `ExecuteL()` is called again, with the original request result in `status`. If `ExecuteL()` starts another asynchronous request, it must return an ''uncomplete'' result again. Otherwise, it should return the function result.

8. If the request result was any other code, the module's `Cancel()` function is called, and the result is converted into an **m** exception, going through `GetErrorMessage()` to check for a module specific exception text.

`Cancel()` is not only called in the case of errors, but also if the process is stopped or closed by the user, if the **m** application is exited, or if a timeout expires. As a consequence, you **must** override `NativeModule::Cancel()` (which does nothing by default) if any of your functions start asynchronous requests.

All this sounds more complicated than it really is: most of the time, observing the following three items is sufficient when implementing an asynchronous function:

- Have your function issue the asynchronous request, passing `status`, and return a value which was `SetUncomplete()`.
- Maintain an internal state so you know whether an asynchronous function is to be started or completes each time `ExecuteL()` is called.

- Override `Cancel()` to cancel the request and to reset the internal state.

Alarm Server Example, Continued

The function `alarm.wait` is asynchronous by its very nature. The alarm server provides function `RASCliSession::NotifyChange()` to asynchronously wait for a change event in the alarm server. The event we are interested in is `EAlarmChangeEventTimerExpired`. All other events should immediately start another wait for change notification. In this case, the request status serves as our internal state which allows us to know whether to start or complete a request:

```
case WaitFunction:
  // only end waiting and return an alarm id if it expired
  if (status == EAlarmChangeEventTimerExpired)
    result.SetNumber(awaitedId);
  // otherwise start or continue waiting
  else {
    // wait for an alarm change notification
    session.NotifyChange(status, awaitedId);
    // signal to the runtime system that this call is asynchronous
    result.SetUncomplete();
  }
  break;
```

Since the alarm id passed to `NotifyChange()` must live between requests, we declare it as a (private) module class member variable:

```
TAlarmId awaitedId;
```

Cancelling the `NotifyChange` request is simple:

```
void Cancel() { session.NotifyChangeCancel(); }
```

If we compile[2] and install this module and try `alarm.wait`, we will get an exception like `[6]: Unknown error`. What's wrong?

The problem is that `NotifyChange()` is somewhat peculiar: on successful completion, it does not assign `KErrNone` to the request status, but a positive `TAlarmChangeEvent` enumeration value. The **m** runtime system treats this as an exception. We therefore must override `HandleError()`

[2]The required library `alarmshared.lib` is missing from most S60 SDKs, but you can safely copy the file from e.g. the UIQ SDK in this case.

to map these positive values to `KErrNone`:

```
TInt HandleError(TInt error) {
  // handle the non-zero positive returns from session.NotifyChange
  return error > 0 ? KErrNone : error;
}
```

After this addition, `alarm.wait` works as expected, except that our specification includes a millisecond timeout.

5.3.1 Timeouts

Adding a timeout to an asynchronous function is simple: the runtime system maintains an internal timer and offers a single function to start it:

- `void AddTimeoutL(TTimeIntervalMicroSeconds32 timeout);`
  Adds a microsecond timeout to an asynchronous request. If the timeout is reached, the native module's Cancel() function will be called, and the request completes with `KErrTimedOut`. If `timeout` is negative, no timeout is added.
  Use `Runtime::Value::GetTimeoutL()` to convert a millisecond number to a microsecond timeout value.

If you do not want your function to throw an exception if the timeout expires, you must handle `KErrTimedOut` in `HandleError()`.

Alarm Server Example, Continued

If `alarm.wait` was called with a parameter, we call `AddTimeoutL()`, passing the parameter value:

```
case WaitFunction:
    ...
    // if the wait included a timeout, add it
    if (paramCount > 0) runtime->AddTimeoutL(params[0].GetTimeoutL());
    ...
```

If we now call `alarm.wait` with a three second timeout, we will most likely get a timeout:

```
alarm.wait(3000)
→ ErrTimedOut thrown
```

We did not specify what should happen if the timeout expires. If we want `alarm.wait` to return `null` instead of throwing an exception, we must override `HandleError()` to map `KErrTimedOut` to `KErrNone`, and introduce a variable to remember it was a timeout the next time `ExecuteL()` is called:

```
TBool isTimeout;
```

```
TInt HandleError(TInt error) {
  // handle timeout
  if (error == KErrTimedOut) { isTimeout = ETrue; return KErrNone; }
  // handle the non-zero positive returns from session.NotifyChange
  else return error > 0 ? KErrNone : error;
}
```

The implementation of `WaitFunction` also requires a small modification:

```
  // if we ended with a timeout, reset flag and return default null
  if (isTimeout) isTimeout = EFalse;
  // only end waiting and return an alarm id if it expired
  else if (status == EAlarmChangeEventTimerExpired)
    ...
```

5.4 Native Objects

The objects produced by native modules are not restricted to **m** types. For instance, the streams returned by `io.open` (Library, p. 50) are *native objects* which cannot be manipulated from within **m**, only by module `io` and a few other modules.

But native objects are not restricted to streams: an instance of any subclass of `CBase` can be wrapped into a native object. When doing so, the instance will automatically be deleted if the native object containing it is freed by garbage collection.

Each native object has a *type* which is a zero terminated character string. The type is used to check for a particular object type (C++ runtime types are not supported by Symbian), and when converting the object to a string, e.g. when printing it to the console. For instance, native streams have the type

```
/**
   The type of all native I/O streams.
*/
static const char * const TYPE = "stream";
```

Native objects are created by the following function from `class Runtime`:

- `Runtime::Value NewNativeL(CBase *obj, const char *type);`
  Creates a new native object. The object `obj` must not be on the cleanup stack, and is destroyed when this function leaves.

To access native objects, `Runtime::Value` offers the following methods:

- `TBool IsNative() const;`
  Checks whether this value is a native object.
- `NativeObject &GetNativeL() const;`
  Gets this value as a native object, leaving if it isn't a native object.
- `CBase *GetNativeObjL(const char *type) const;`
  Gets the object defined by this native object. Leaves if this isn't a native object, or if it has a different type.

And here is a sample fragment using native objects:

```
class MyObject : public CBase {
public:
  static const char * const TYPE = "myobject";
  static MyObject *NewL();
  ...
  TInt count() const;
}

Runtime::Value ExecuteL(TInt index, Runtime::Value *params,
                        TInt paramCount, TRequestStatus &status)
{
  MyObject *obj;
  Runtime::Value result;
  switch (index) {
  case NewFunction:
    // create a native object around a MyObject instance
    obj = MyObject::NewL();
    result = runtime->NewNativeL(obj, MyObject::TYPE);
    break;
  case CountFunction:
    // return the result of obj->count()
    obj = (MyObject*)params[0].GetNativeObjL(MyObject::TYPE);
    result.SetNumber(obj->count());
    break;
  ...
```

5.5 Native Streams

A special class of native objects are native streams. These are subclasses of `class NativeStream`, declared in `NativeStream.h`. Once created, instances of native streams are accessed by module `io` (Library, p. 43).

When subclassing `NativeStream`, you **must** implement the following abstract virtual functions:

- `virtual TBool ReadL(TDes8 &buf, TRequestStatus &status);`
  The raw read method. If the read occurs at EOF, `buf.Length()` must be set to zero. Returns `ETrue` if the call was asynchronous, `EFalse` if the data was already available.

- `virtual TBool WriteL(const TDesC8 &buf, TRequestStatus &status);`
  The raw write method. All `buf.Length()` bytes must be written (not necessarily in a single call). Returns `ETrue` if the call was asynchronous, `EFalse` if the data was already written.

If the above methods ever return `ETrue`, the following method must also be overridden:

- `virtual void Cancel();`
  Cancel an asynchronous I/O operation.

In addition, the class **can** override the virtual functions `SeekL()`, `Close()`, `SizeL()`, `AvailL()`, `HandleError()`.

Here is a sample implementation of a native stream subclass reading from and writing to a socket:

```
class MyStream : public NativeStream {
  RSocket *socket; // owned by this class
  TSockXfrLength xfrLength;

  ~MyStream() {
    // garbage collecting the stream should also delete the socket
    delete socket;
  }
public:
  static MyStream *NewLC(RSocket *socket) {
    MyStream *self = new (ELeave) MyStream;
    CleanupStack::PushL(self);
    self->socket = socket;
    self->ConstructL();
    return self;
  }

protected: // from NativeStream
  TInt Close() {
    socket->Close();
    return NativeStream::Close();
  }

  TBool ReadL(TDes8 &buf, TRequestStatus &status) {
    socket->RecvOneOrMore(buf, 0, status, xfrLength);
    return ETrue; // was asynchronous
  }

  TBool WriteL(const TDesC8 &buf, TRequestStatus &status) {
    socket->Write(buf, status);
    return ETrue; // was asynchronous
  }

  void Cancel() {
    NativeStream::Cancel();
    socket->CancelAll();
  }
};
```

5.6 Native Icons

Another special class of native objects are native icons. These are instances of `class NativeIcon`, declared in `NativeIcon.h`. Native icons are created and used by module `graph` (Library, p. 73).

Native icons are simply containers for images stored in memory. A native icon owns one or two associated bitmaps (pixel maps) of type `CFbsBitmap`, one for the image and an optional one for the mask. If present, the mask must have the same size as the image, and display mode `EGray2`.

If you write image processing software, it should ideally operate on `NativeIcon` instances, so it can be used directly by module `graph`.

For instance, to access the image map of a native icon stored in an **m** value:

```
Runtime::Value v = ...
CFbsBitmap *bitmap =
  ((NativeIcon*)v.GetNativeObjL(NativeIcon::TYPE))->GetImageMap();
```

To create a native icon from a bitmap:

```
CFbsBitmap *bitmap = ...
CleanupStack::PushL(bitmap);
NativeIcon *icon = new(ELeave) NativeIcon(bitmap);
CleanupStack::Pop(); // bitmap
// NewNativeL() destroys the object if it leaves, so it must not
// be on the cleanup stack
Runtime::Value v = runtime->NewNativeL(icon, NativeIcon::TYPE);
```

5.7 Memory Allocation

All memory allocated via `runtime->New...L()` calls is subject to garbage collection: if it is not referenced by an **m** variable (or array), it will be collected and freed. The **m** runtime system maintains references to all memory chunks it has allocated. **m** objects must therefore never be pushed on the Symbian cleanup stack.

No garbage collection takes place during native function calls, or between calls to `ExecuteL()` when waiting for asynchronous events.

As a side effect of memory allocation and garbage collection, the location

of variables in memory may change *between* calls to native functions. C++ pointers and references to **m** values are therefore only valid during a call to `ExecuteL()`.

If your module creates Symbian objects or allocates raw memory outside the **m** universe and does not tie the object or memory to an **m** native object value, you must ensure yourself it is properly freed when the module is destroyed (i.e. in its destructor). Closing an **m** process will destroy all its module instances.

5.8 Module Loading and Initialization

When the **m** compiler processes a `use` clause, it first checks whether a module has already been loaded. If it has been loaded, the compiler simply continues. If it has not been loaded, the following happens:

1. The compiler searches for a DLL with name *`module`*`_mm.dll`
2. If there is no such DLL, it attempts to load the source of a module written in **m**.
3. If there is a DLL, it attempts to load it.
4. It then calls the DLL's first exported function, assuming the returned object being an implementation of the abstract `class NativeModule`.
5. It checks some magic tag and the runtime version expected by the module.
6. It assigns a reference to an implementation of `class Runtime` to the module's `runtime` member.
7. It calls the module's second phase constructor `ConstructL()`.

Any error in each of the above phases causes compilation to be aborted (if the `use` clause specified a `try`, an error results in an empty module stub with version `null`).

Since errors during loading and initialization cannot be catched or handled by an **m** script, any module specific initialization like connecting

to servers, opening files and such should be delayed until the first call of `ExecuteL()` whenever possible. This also avoids allocating resources if no functions of the module are ever going to be called.

5.9 Restrictions

Native functions in **m** must observe a few restrictions:

- There is no way to call an **m** function from within a native function. It can only create **m** values and call C++ functions. For instance, if your module needs to call an **m** function as a callback, you must implement the callback mechanism in an **m** wrapper module calling your native module.
- Native functions execute from within an active object (`CActive::RunL()` function). All **m** processes share the same thread. As a consequence, all other **m** processes must wait until a native function returns. Make sure your native function does not do lengthy computations, or does it in small chunks.
- The **m** runtime system is **not** thread safe. If your native module creates its own thread, you must make sure only the **m** thread calls the runtime system, and provide appropriate synchronization.

A. NMI SDK

Writing native modules for **m** requires the NMI SDK in addition to a full Symbian OS SDK. The NMI SDK is distributed as a ZIP file, without explicit installer.

Installation

Unpack `nmisdk.zip` into a directory of your choice. On Windows, a good option is the `C:\Symbian` directory used as default by most Symbian SDK installers. The NMI SDK will then reside in `C:\Symbian\nmisdk`. For most standard SDKs, you should probably also copy some of

- `nmisdk\lib\S60\mRuntime.lib` (for S60 2nd Ed)
- `nmisdk\lib\S60_3\mRuntime.dso` (for S60 3rd Ed, self signed and DevCert signed)
- `nmisdk\lib\S60_3-OS\mRuntime.dso` (for S60 3rd Ed, online signed)
- `nmisdk\lib\S60_5\mRuntime.dso` (for S60 5th Ed, self signed and DevCert signed)
- `nmisdk\lib\S60_5-OS\mRuntime.dso` (for S60 5th Ed, online signed)
- `nmisdk\lib\UIQ\mRuntime.lib` (for UIQ)
- `nmisdk\lib\UIQ3\mRuntime.dso` (for UIQ3, self signed and DevCert signed)
- `nmisdk\lib\UIQ3-OS\mRuntime.dso` (for UIQ3, online signed)

into `Epoc32\release\armi\urel` of your SDK installation(s).

Files

The NMI SDK contains the following files and directories:

- `doc/nmi.pdf`: this document.
- `examples/Alarm`: the alarm server example.
- `examples/World`: the world server example.
- `include/ErrorCodes.h`: the additional **m** error codes (exception tags).
- `include/NativeIcon.h`: the header file for native icons.
- `include/NativeModule.h`: the header file for native modules.
- `include/NativeStream.h`: the header file for native I/O streams.
- `include/Runtime.h`: the header file and reference for the runtime system.
- `include/TextWriter.h`: a simple interface to write to the **m** process console.
- `include/UserPermissions.h`: the **m** permission flags to check.
- `lib/S60/mRuntime.lib`: the **m** runtime link library for S60 2nd Edition.
- `lib/S60_3/mRuntime.dso`: the **m** runtime link library for S60 3rd Edition, SID of self signed and DevCert signed **m** runtime DLLs.
- `lib/S60_3-OS/mRuntime.dso`: the **m** runtime link library for S60 3rd Edition, SID of online signed **m** runtime DLLs.
- `lib/S60_5/mRuntime.dso`: the **m** runtime link library for S60 5th Edition, SID of self signed and DevCert signed **m** runtime DLLs.
- `lib/S60_5-OS/mRuntime.dso`: the **m** runtime link library for S60 5th Edition, SID of online signed **m** runtime DLLs.
- `lib/UIQ/mRuntime.lib`: the **m** runtime link library for UIQ.

- `lib/UIQ3/mRuntime.dso`: the **m** runtime link library for UIQ3, SID of self signed and DevCert signed **m** runtime DLLs.
- `lib/UIQ3-OS/mRuntime.dso`: the **m** runtime link library for UIQ3, SID of online signed **m** runtime DLLs.
- `m/dllinstall.m`: an **m** script helping in installing `.dll` files transferred to the device (Symbian 2nd Edition only).

B. Platform Security

With the 3rd edition of its OS, Symbian OS has introduced *platform security*, complicating matters for developers.

m for Symbian 3rd Edition thus comes in three variants (see also chapter 5 (Reference, p. 67)):

- A self signed package with the following capabilities:

```
LocalServices NetworkServices ReadUserData UserEnvironment
WriteUserData
```

- An unsigned package for online signing, with the following capabilities:

```
LocalServices NetworkServices ReadUserData UserEnvironment
WriteUserData Location PowerMgmt ProtServ ReadDeviceData
SurroundingsDD SwEvent TrustedUI WriteDeviceData
```

- An unsigned package for signing with a Developer Certificate (DevCert), with the following capabilities:

```
LocalServices NetworkServices ReadUserData UserEnvironment
WriteUserData Location PowerMgmt ProtServ ReadDeviceData
SurroundingsDD SwEvent TrustedUI WriteDeviceData
CommDD DiskAdmin MultimediaDD NetworkControl
```

In order to be loadable by an application, a DLL must have at least the capabilities of the process loading it (the capabilities of a process never change during its lifetime). This has the following consequences[1]:

- You must compile your code with at least the capabilities listed above.

[1] There are tools to modify the capabilities of the executables in a SIS file and re-sign it, gaining complete flexibility over capabilities (within the limits of platform security, of course). You should however assume that other **m** installations will run with either of the capabilities listed here.

- Adding more capabilities than those of the installed **m** package will have no effect.
- Signing types of DLL and **m** installation cannot be freely mixed. The following matrix shows the possible combinations:

Signing Type	Self DLL	Online DLL	DevCert DLL
Self **m**	yes	no	yes
Online **m**	no	yes	no
DevCert **m**	no	no	yes

If you plan to make your library publicly available via SIS packages, make sure you provide all three packages (self signed, online signable unsigned, DevCert signable unsigned) with the corresponding capabilities.

Index

active objects, 31
AddConstantL, 10
AddNativeFunctionL, 9, 11, 18
AddTimeoutL, 34
AlarmModule.cpp, 27
Array, 18
asynchronous requests, 31

bld.inf, 24
BooleanValue, 16

CActive, 31
CActive::RunL(), 41
Cancel, 7, 32
capabilities, 25, 47
CBase, 35
CheckPermissionsL, 30
ConstructL, 6, 9

DevCert signed, 23, 24, 43--45
DLL, 3
    entry points, 12
    structure, 5
dynamic link library, 3

edll.lib, 24
EKA2, 12
euser.dso, 24
euser.lib, 24
ExcNotPermitted, 30
ExecuteL, 6, 11
ExpectedRuntimeVersion, 6

garbage collection, 39
GetArrayL, 16
GetBooleanL, 16
GetCurrentCityL, 20
GetErrorMessage, 7, 32
GetIntL, 16
GetNativeL, 36
GetNativeObjL, 36
GetNumberL, 16
GetPtrCL, 16
GetStringL, 16
GetTimeL, 16, 29
GetTimeoutL, 16, 34
GetUidL, 16
GetVariableL, 9, 10

HandleError, 7, 32, 33
HandleProcessEventL, 7

IsNative, 36
IsNull, 16
IsNumber, 16
IsString, 16

KErrTimedOut, 34

ModuleVersion, 7, 9
mRuntime.dso, 24

mRuntime.lib, 24

native icons, 39
Native Module Interface, 3
native objects, 35
native streams, 37
NativeIcon.h, 39
NativeModule, 6
NativeStream.h, 37
NewArrayL, 16, 18
NewNativeL, 36
NewStringL, 16, 17
NMI, 3

online signed, 23, 24, 43--45

package file, 26
platform security, 47

RASCliSession, 27
Runtime, 9, 15, 30, 36
runtime, 8
runtime system, 15
Runtime.h, 15
Runtime::Array, 15, 20
Runtime::Array::GetValues, 19
Runtime::Array::SetL, 19
Runtime::NativeObject, 15
Runtime::String, 15, 17
Runtime::Value, 10, 15, 16, 36
Runtime::ValueArray, 15, 19, 20
Runtime::VERSION, 9
RWorldServer, 5

SDK, 43
    files, 44
self signed, 23, 25, 43--45, 47
SetBoolean, 16
SetNull, 16
SetNumber, 16
SetTime, 16
SetTimeout, 16
SetUncomplete, 32
SIS package, 26
String, 17

t32wld.h, 8
TASShdAlarm, 29
thread safe, 41
timeout, 34
TRequestStatus, 31
TUid, 16

UID, 26
UID2, 23
UID3, 23
user permissions, 30
UserPermissions.h, 30

World.mmp, 24
WorldModule.cpp, 5