Simple Device Model

User Manual

Version 1.0.3

Microproject LLC

Contents

Chapter 1

Introduction

1.1 Overview

Simple Device Model (SDM) is a platform designed to interact with hardware. Its typical applications include instrument control and data acquisition. SDM employs an abstraction layer which makes the device accessible via a standard API exposing the device's address space and data streams in a hardware-independent manner. The actual device-specific code is encapsulated in a plugin.

SDM framework is a set of tools and libraries to work with devices via SDM plugins. SDM framework provides both interactive tools and a scripting engine. It also includes an SDK which contains everything needed to develop a plugin and example plugins.

SDM distribution can be also shipped with additional plugins to work with the specific devices. These plugins are not considered parts of the SDM framework.

1.2 Structure of this manual

System requirements, procedures for installation and building SDM from source are covered in Chapter 2, *Installation*.

Chapter 3, *Basic usage*, contains basic information needed to use SDM tools.

Chapter 4 (*Register map window*) and Chapter 5 (*Plotter window*) describe `sdmconsole` windows used to control devices and display received data.

Chapter 6 describes the scripting engine provided by the SDM framework. The engine is based on Lua programming language and contains extensions for both device access and user interaction.

Chapter 7, *Creating SDM plugins*, covers the SDM plugin interface and the procedure of plugin development.

Chapter 9, *SDM API reference*, provides a detailed description of SDM plugin API functions.

Chapter 2

Installation

2.1 System requirements

SDM is officially supported on machines with an x86 or x86-64 CPU architecture and one of the following operating systems:

- Microsoft® Windows® 7 SP1 or later;

- Ubuntu® 18.04 or later.

It is also possible that SDM will work on other platforms. In particular, any modern (vendor-supported) Linux® distribution that can install the build system dependencies (Subsection 2.4.1) is likely capable of building and running SDM.

2.2 Installation on Microsoft Windows

Under Microsoft Windows, the recommended way to install SDM is to use the provided installer package. It requires administrator privileges to run. The following command line switches are supported:

- `/S` – Perform a silent installation/uninstallation.

- `/D=`*path* – Override installation path. Must be the last argument. Can be used for both silent and non-silent modes. *path* must not include any quotes, even if it contains spaces.

Note that while SDM itself is Unicode-aware, Lua interpreter is not. For that reason it is strongly recommended that the installation path doesn't contain characters not present in the current ANSI code page.

Uninstallation is performed, as usual, from the Control Panel or by running `uninstall.exe` directly.

2.3   Installation on Ubuntu

Installation is performed by building SDM from source. First, build system dependencies must be installed:

```
sudo apt install build-essential qtbase5-dev qtbase5-dev-tools \
    libqt5svg5-dev qttools5-dev qttools5-dev-tools \
    qttranslations5-l10n qt5-image-formats-plugins cmake
```

After that, SDM itself can be built with the following commands (assuming that they are run from the SDM source distribution top-level directory):

```
mkdir build
cd build
cmake ../src
make -j
sudo make install
```

SDM can be removed from the system by running `uninstall.sh` from the build directory. See also Section 2.4 for detailed build instructions.

2.4   Building from source

2.4.1   Build system dependencies

- A toolchain for C++ development:

    - for Microsoft Windows: Microsoft Visual Studio® 2013 or later, or MinGW-w64 GCC 4.8 or later (see notes below). Pre-built Windows packages are created with Microsoft Visual Studio 2013.

    - for Linux: GCC 4.9 or later, or Clang 3.4 or later.

- CMake 3.3 or later;

- Qt® 5.x, $x \geq 6$ (optional, for `sdmconsole`);

- NSIS 3.0 or later (optional, for Windows installer);

- TeX Live 2013 or later (optional, for LaTeX documentation).

All tool locations must be added to the `PATH` environment variable.

Notes on MinGW-w64 toolchains

The MinGW-w64 project provides multiple flavors of GCC toolchains differing in threading and exception handling models. To build SDM, *posix* threading model is required; SDM doesn't depend on a particular exception handling model, although *sjlj* is recommended for x86 targets and *seh* – for x86-64 targets.

2.4.2 Build procedure

SDM uses CMake as a build system generator. CMake automatically generates scripts for a native build system (either a set of Makefiles, or an IDE project), which in turn performs the actual build. The following build system generators have been tested to work with SDM:

- For Microsoft Windows:

  - `"MinGW Makefiles"` (Makefile-based, for MinGW)
  - `"MSYS Makefiles"` (Makefile-based, for MinGW+MSYS)
  - `"NMake Makefiles"` (Makefile-based, for Visual Studio)
  - `"Visual Studio 12 2013"` (IDE-based)
  - `"Visual Studio 14 2015"` (IDE-based)

- For Linux: `"Unix Makefiles"` (active by default, can be omitted)

- For all platforms: `"Ninja"`

Build system generator is not necessarily tied to a specific compiler. For Makefile-based generators (and Ninja) you can set the compiler with the `-DCMAKE_C_COMPILER` and `-DCMAKE_CXX_COMPILER` CMake options (see also Subsection 2.4.3).

To build SDM, the following steps must be performed:

1. Create a build directory outside of the source tree.

2. Run CMake from the build directory:

```
cmake [ -G generator ] [ options ] source_directory
```

Possible *generator* values are provided above. To build a 64-bit version of SDM with Visual Studio IDE-based generators, add `-A x64` option. For a list of other options see Subsection 2.4.3.

3. Build the source code. For Makefile-based generators, a *make* command must be invoked. The name of *make* command depends on the toolchain: it is usually `make` for Linux; for Microsoft Windows in can be `nmake` for Microsoft Visual Studio, `make` for MSYS and `mingw32-make` for MinGW. Alternatively, the third-party `jom` tool[1] can be used in place of `nmake` or `mingw32-make`.

   For Visual Studio generators, build the `ALL_BUILD` target.

4. To install the project, invoke *make* `install` for Makefile-based generators or build the `INSTALL` target for Visual Studio generators.

5. Optionally, one can run the SDM test suite by invoking *make* `test` for Makefile-based generators or building the `RUN_TESTS` target for Visual Studio generators.

The CMake-based build system won't build the Windows installer and this manual. Building the installer is described in Subsection 2.4.5 below. The user manual is built with `lualatex`.

2.4.3   Build options and variables

Additional build options can be passed to CMake using `-D` *option=value* syntax. The most important options are listed in table 2.1.

On Microsoft Windows the default value of the `CMAKE_INSTALL_PREFIX` variable is `C:/Program Files/SDM`. It is used only by the CMake `install` target and doesn't affect compilation.

The default `CMAKE_INSTALL_PREFIX` on Linux is usually `/usr/local` and should be set to the correct path for the software to be able to detect paths to data directories (SDM will try to determine installation path at run time, but Linux lacks a guaranteed portable way to do so).

The `CMAKE_BUILD_TYPE` option affects only Makefile-based generators. For Visual Studio generators, CMake creates a solution with multiple configurations that can be switched from the IDE itself.

---

[1] `jom` is an `nmake` clone developed by the Qt project. Unlike `nmake`, it supports parallel builds. See `https://wiki.qt.io/Jom`.

Table 2.1: SDM build options and variables

Option	Description
General CMake options	
CMAKE_INSTALL_PREFIX	Installation prefix
CMAKE_BUILD_TYPE	Build type. Values: Release, Debug, MinSizeRel, RelWithDebInfo; default value depends on a chosen generator.
CMAKE_PREFIX_PATH	Additional paths to search for dependencies
CMAKE_C_COMPILER	C language compiler
CMAKE_CXX_COMPILER	C++ language compiler
CMAKE_RC_COMPILER	Resource compiler (Microsoft Windows only)
SDM specific options (ON/OFF, default is OFF unless otherwise noted)	
OPTION_DIAGNOSTIC	Employ additional compile-time and run-time diagnostics, some of which can be inappropriate for released packages.
OPTION_LUA_SYSTEM	Link against the Lua library installed on this system (default is to use the version bundled with SDM). Lua 5.3 or later is required.
OPTION_NO_LUAMODULES	Skip building optional Lua modules.
OPTION_NO_QT	Don't use Qt. Targets dependent on Qt (such as sdmconsole) won't be built.
OPTION_USE_VCREDIST	When building the installer package, deploy the Microsoft Visual C++ Runtime using vcredist_*.exe (MSVC only). Default is OFF for Microsoft Visual Studio 2013, ON for later versions. See also Subsection 2.4.5.
OPTION_VALGRIND	Enable using valgrind by the test suite to track memory leaks (Linux only).
OPTION_NO_ICONS	Skip installation of desktop entries (Linux only).
SDM specific variables	
VCREDIST_PREFIX	Directory to search for the Microsoft Visual C++ Redistributable package (MSVC only, optional, autodetected by default).

The `CMAKE_PREFIX_PATH` option is used to specify paths to dependencies installed to non-standard locations (e.g. `/opt`). Alternatively, one can create an environment variable with the same name. The `PATH` environment variable can also be used for this purpose.

When linking SDM against an external Lua installation, note that SDM and the Lua interpreter should use the same instance of the Standard C library, otherwise certain features (e.g. passing file handles) will not work properly. Using `OPTION_LUA_SYSTEM` on Microsoft Windows is thus not recommended unless you can ensure that it is the case; when in doubt, use the bundled Lua interpreter. This should not be a problem for Linux.

2.4.4   Examples

A few examples of building SDM using Makefile-based generators are provided below. The examples are assumed to be run from the SDM source distribution top-level directory.

For Linux:

```
mkdir build
cd build
cmake ../src
make -j
sudo make install
```

For Microsoft Visual Studio:

```
mkdir build
cd build
cmake -G "NMake Makefiles" ../src
nmake
nmake install
```

For MinGW:

```
mkdir build
cd build
cmake -G "MinGW Makefiles" ../src
mingw32-make -j
mingw32-make install
```

For MSYS:

```
mkdir build
cd build
cmake -G "MSYS Makefiles" ../src
make -j
make install
```

2.4.5 Generating installer for Windows

Under Microsoft Windows, SDM provides a script, `make_installer.nsi`, which can be used to generate an installer with NSIS (Nullsoft Scriptable Install System)[1].

To generate the installer package, SDM must be first built and installed using CMake (possibly to a temporary directory). The NSIS script is configured by the build system and installed alongside program executables. Then, `makensis` program is used to create an installer.

The installer generation script uses the `EnVar` NSIS plugin to modify the PATH environment variable in the Windows Registry. This plugin is not included in the NSIS distribution and must be installed separately[2].

When using a MinGW-based toolchain, it is recommended to use `make install/strip` instead of just `make install` to strip unneeded symbols from the executables:

```
mkdir build
cd build
cmake -G "MinGW Makefiles" -DCMAKE_INSTALL_PREFIX=tmp ../src
mingw32-make -j
mingw32-make install/strip
cd tmp
makensis make_installer.nsi
```

External DLL dependencies

To run SDM, a target system must have shared versions of the Standard C++ library and Qt libraries installed. On Microsoft Windows, the `install` target will run a helper tool (`dlldeptool.exe`) that automatically copies the required libraries from the host system to the application directory. If you plan to distribute the produced package, ensure that you are allowed

---

[1] `http://nsis.sourceforge.io`
[2] `https://nsis.sourceforge.io/EnVar_plug-in`

to redistribute these libraries. Notably, debug versions of Microsoft Visual
C++ Runtime are *not* redistributable.

In addition to the above, since Microsoft Visual Studio 2015 compiled
binaries require the *Universal CRT* to be present on the target system. This
component is shipped with Microsoft Windows 10; previous versions can get
it via the Windows Update. The required update package (KB2999226) can
also be installed manually[1]. Alternatively, it is possible to deploy the stan-
dard libraries centrally using `vcredist_x86.exe` (or `vcredist_x64.exe`),
as described in the Subsection 2.4.3; these packages also deploy the *Uni-
versal CRT* if it is not already present. In this case the standard libraries
are not copied to the application directory. This option is active by default
when using Microsoft Visual Studio 2015 or later.

2.4.6   Cross compilation

It is possible to build Windows binaries on a Linux host using MinGW-w64
cross compiler toolchain. To do so, some additional CMake variables must
be set (replace `i686` with `x86_64` to build 64-bit executables):

```
cmake \
    -DCMAKE_SYSTEM_NAME=Windows \
    -DCMAKE_C_COMPILER=i686-w64-mingw32-gcc \
    -DCMAKE_CXX_COMPILER=i686-w64-mingw32-g++ \
    -DCMAKE_RC_COMPILER=i686-w64-mingw32-windres \
    ../src
```

To build `sdmconsole`, a cross-compiled version of Qt must be present
and available to CMake (the `CMAKE_PREFIX_PATH` option can be used to
specify its location). `install` and `test` targets require Wine to be installed.
`PATH` environment variable in Wine must contain MinGW runtime DLL
locations. Note that `PATH` environment variable in Wine is not related to the
host environment variable with the same name. In order to set it, create
a `PATH` parameter in the `HKEY_CURRENT_USER\Environment` registry key
using `wine regedit`. Directory names must not contain trailing slashes.

2.5   Directory hierarchy

Installation directories for SDM components are listed in Table 2.2. Paths
are relative to the main installation directory.

---

[1]`https://www.microsoft.com/en-us/download/details.aspx?id=48234`

Table 2.2: SDM directory hierarchy

| Directory | | Purpose |
Windows	Linux	
`.`	`bin`	Program executables
`include`	`include/sdm`	Development headers
`lib`	`lib/sdm`	Development libraries
`lua`	`lib/sdm/lua`	Lua extension modules
`plugins`	`lib/sdm/plugins`	SDM plugins
`qt`	—	Qt plugins (Windows)
`cmake`	`share/sdm/cmake`	CMake package metadata
`data`	`share/sdm/data`	SDM plugin data
`doc`	`share/sdm/doc`	Documentation
`examples`	`share/sdm/examples`	Example SDM plugins
`scripts`	`share/sdm/scripts`	User-visible Lua scripts
`translations`	`share/sdm/translations`	Translation files

User settings are stored in `%APPDATA%\Microproject` on Microsoft Windows and in `$HOME/.config/Microproject` on Linux.

On Linux, how desktop entries (shortcuts and icons) are installed depends on the `CMAKE_INSTALL_PREFIX` value:

- if `CMAKE_INSTALL_PREFIX` is in the current user's home directory, desktop entries are installed to `$HOME/.local/share`;

- otherwise, if `CMAKE_INSTALL_PREFIX` is `/usr`, desktop entries are installed to `/usr/share`;

- otherwise, `/usr/local/share` is used.

Chapter 3

Basic usage

SDM framework includes two client applications that can be used to manage SDM devices:

- `sdmconsole`: a GUI application providing graphical interactive tools and a scripting API based on Lua.

- `sdmhost`: an application with a command line interface that can be used to execute scripts, both in batch mode (reading from file) and interactively (reading commands line by line from the standard input).

`sdmconsole` and `sdmhost` use the same scripting engine.

A headless version of `sdmhost` (called `sdmhostw`) is also provided under Microsoft Windows; unlike the regular `sdmhost`, the alternative version doesn't use a console window. Platforms other than Windows don't distinguish between console and GUI applications and therefore don't need this version.

This chapter will focus on `sdmconsole` since `sdmhost` implements a subset of its features.

Command line syntax for `sdmconsole` and `sdmhost` is described in Appendix A.

3.1 Plugin installation

SDM plugins are implemented as shared libraries (`.dll` on Windows, `.so` on Linux). It is recommended that plugins are put into the SDM plugins directory (Section 2.5), although loading plugins from other locations is also possible.

Under Microsoft Windows, when a plugin has DLL dependencies that are not used by SDM itself, the required DLL files must be either installed system-wide or copied to the main SDM installation directory alongside the program executables (not the `plugins` subdirectory).

3.2   Basic concepts: devices, channels, sources

SDM uses a tree-like hierarchy of objects to access devices. On the top level there are *plugin* objects representing SDM plugins. A plugin object can be a parent to one or several *device* objects which in turn can have *channel* and *source* objects as children. Features provided by these objects are summarized below:

- *Plugin*: open device objects.

- *Device*: connect/disconnect, query connection status, open channel and source objects.

- *Channel*: access the device's address space (write/read registers and memory blocks).

- *Source*: capture data streams from the device.

Each SDM object, regardless of type, can also have *properties*, some of which are immutable (e.g. device name) and some are writable (e.g. connection settings).

3.3   `sdmconsole` interface overview

The main window of `sdmconsole` (figure 3.1) consists of two parts: the *sidebar* on the left side, and the *dock area* on the right side. Distribution of space can be adjusted by dragging the border between them.

The upper part of the sidebar displays the object tree. The lower part contains a set of controls to manage the currently selected object and view or edit its properties.

The dock area is able to contain an arbitrary number of secondary windows. Secondary windows in the dock area can be freely resized and moved by the user. They can also be undocked (detached from the dock area) and docked again. Undocked windows are floating on the desktop as any other top-level window. The *View* menu provides options to control how new secondary windows are added to the dock area.
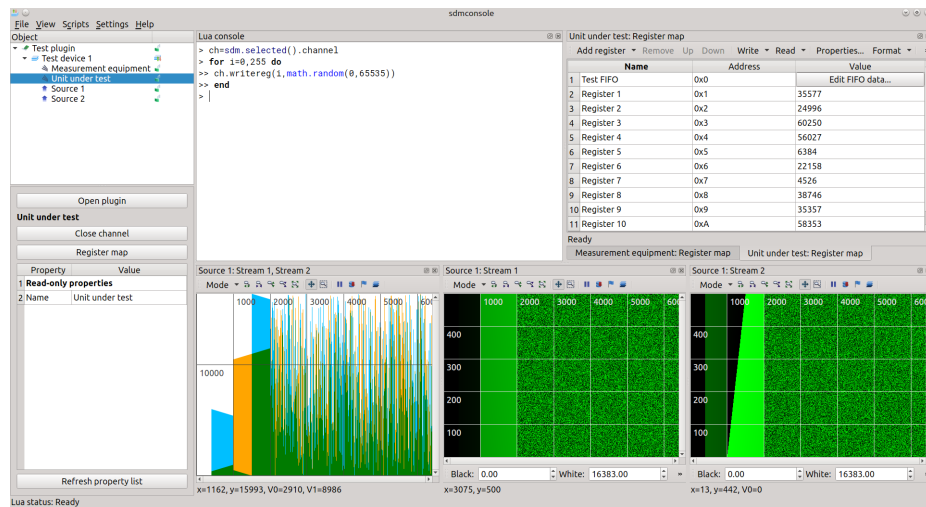
Figure 3.1: `sdmconsole` main window

On the program startup, the dock area has only one window: the *Lua console* used to execute Lua scripting commands. When closed, the Lua console is not destroyed, just hidden; it can be restored using the *View* menu.

3.4   Basic device management

Operations with object tree items are performed using controls in the lower part of the sidebar or context menus activated by right-clicking the object tree. Plugins are loaded with the *Open plugin* sidebar button, the respective context menu item, or the *File → Open plugin…* main menu item. Invoking this command produces a dialog window where the user can select a plugin and a device to load. Alternatively, the device can be opened later, by selecting the plugin in the object tree and clicking on the *Open device* button. Generally, SDM allows the user to open the same device multiple times (unless prevented by a plugin), which can be useful if several devices are connected to the same workstation.

Selecting a device in the object tree produces a device panel which can be used to connect to the device as well as to open channels and sources. Clicking on the *Connect* button produces a connection dialog which allows the user to enter connection parameters (a set of parameters is specified by the plugin). These commands are also available from the device's context menu.

An object can be closed by selecting it in the object tree and clicking on a respective panel button. Closing the object also closes all of its children.

3.5   Accessing the device's address space

`sdmconsole` provides a *Register map* window which is used to access the device's address space. To open this window, select a channel in the object tree and click on the *Register map* button on the channel panel.

A register map is organized as a set of pages, each of pages being a three-column table. The first column contains an optional name of the register or memory block, the second – its address, the last column holds the register value or a button to access the memory block data. Using toolbar commands, one can add new pages, section headers, registers and memory blocks. Registers and memory blocks can be written and read by using toolbar buttons or F2 and F3 keys respectively.

By default, the user enters register data directly as a numerical value. Using *Properties* button, one can replace the text input field in *Data* column with a drop-down list, a combobox or a push button, to provide a selection of possible values.

Registers and memory blocks can have custom write and/or read actions. A custom action is a Lua script which, if set, is executed on write/read instead of simply writing or reading the register or memory block.

Register map can be saved to a file and loaded from a file.

Detailed description of register map's features is provided in Chapter 4.

3.6   Working with data streams

Data streams are accessed by selecting a source object in the object tree. Clicking on a *Stream viewer* button opens a dialog where the user can choose one or more streams to view. Streams are displayed by a plotter window which allows the user to select different view modes and provides navigation tools. Plotter window is described in detail in Chapter 5.

A stream or several streams can be also saved to a file by clicking on the *File writer* button. Files use the MHDB format specified in Appendix B. The MHDB format requires constant packet length that can be entered by the user or detected automatically. Any longer packet will be trimmed, shorter packets will be padded with zeros.

The user can also set the decimation factor which makes SDM deliver only one of each *N* packets. Decimation factor affects both *Stream viewer* and *File writer*.

3.7 Lua console

Lua console is used to execute Lua statements interactively. Multi-line blocks are also supported. If a statement returns any values, they are printed in the console window; thus, **return** *expression* statement can be used to display the result of *expression* evaluation. For tables, each key/value pair is printed separately; **return** **_G** prints the global environment.

Table 3.1 lists special commands that are executed by the console itself and not passed to the Lua interpreter.

Table 3.1: Special Lua console commands

Command	Description
clear	Clear the console window
quit	Quit the application

Lua console maintains a history of entered commands which can be navigated through using up and down arrow keys. Commands starting with a space are not added to the history.

Script files can be executed using the **dofile**() Lua function, or, if Unicode file name support is desired, codec.dofile() (see Subsection 6.3.1). Script files can also be dropped on the Lua console or selected in the *Scripts* menu.

See also Chapter 6 for detailed description of SDM Lua extensions.

Chapter 4

Register map window

Register map window is a versatile tool provided by `sdmconsole` allowing the user to work with the device address space. It is organized as a set of table-based pages (Figure 4.1).

Figure 4.1: Register map window

To add a page, select the *Add page* menu item which is available after clicking on an arrow near the *Add register* toolbar button. To change page name, double-click on its tab. To remove page, click on the *Close* icon on the tab.

Table rows can be of multiple types:

- *Register* (default) normally represents a single value that can be written or read. A register is usually defined by its address (second column), its value is stored in the third column.

- *FIFO* represents a memory block that is mapped to a single register address (contained in the second column). The third column shows a button which can be used to access memory contents and other options.

- *Memory* represents a memory block that occupies a contiguous address range (starting from the address contained in the second column). As above, memory contents can be accessed using a button in the third column.

- *Section* is just a header. It contains only one cell and can't be written or read.

Rows can be added, removed and moved using respective toolbar buttons. For *Register*, *FIFO* and *Memory* type rows, the *Properties* button opens a dialog with additional settings for this row.

Rows of *Register*, *FIFO* and *Memory* types can be written and read using *Write* and *Read* toolbar buttons or *F2* and *F3* keys respectively. By default, only the selected row is written/read. *Write* and *Read* buttons also have menus that can be used to write/read the whole page (or all pages). There is also an option to write automatically when the register data are changed.

SDM represents register addresses and values as unsigned 32-bit integers (see Section 7.2), though register map can also display them as signed if needed. With the *Format* toolbar button the user can choose number representation for address and data columns (*As is*, *Unsigned*, *Signed* or *Hexadecimal*). Values entered by the user will be automatically converted to the selected representation (unless the *As is* option is selected).

Save and *Load* toolbar buttons are used to save register map contents to a file and load it from a file. All the columns are saved, as well as additional register, FIFO and memory settings.

4.1   *Register* type row

To add a new row of *Register* type, click on the *Add register* toolbar button. For convenience, each page also has an empty row of *Register* type at the bottom.

By default, the third (data) column contains a text input field where the user can enter a value to write and which displays the value that has been read. By clicking on the *Properties* toolbar button, one can replace this input field by a drop-down list, a combobox or a push button (Figure 4.2).

These input widgets operate as follows:
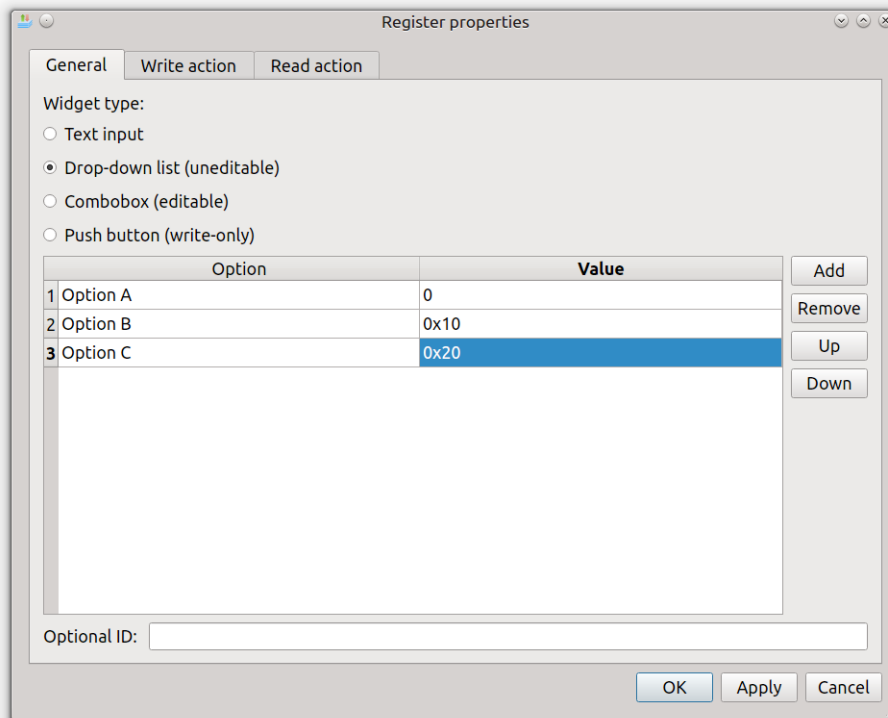
Figure 4.2: *Properties* dialog for *Register* type rows

- A drop-down list allows the user to select an option by name. The value that will be written for this option is specified in the *Properties* dialog. When reading, it will select the option corresponding to the value that has been read. If there is no such option, drop-down list will be converted to a combobox.

- A combobox is similar to a drop-down list, but also allows the user to enter a value directly, bypassing suggested options.

- A push button is used to write a single value that is specified in the *Properties* dialog. Push buttons don't display read values at all.

The user can also define custom actions that are executed when the register is written and/or read instead of default write/read logic (Section 4.3).

4.2   *FIFO* and *Memory* type rows

To add a *FIFO* or *Memory* type row, select an *Add FIFO* or *Add memory* item of the menu attached to the *Add register* toolbar button.

FIFO and Memory type rows work in a similar way, the difference between them is that FIFO represents a memory block mapped to a single register address, whereas memory block occupies a contiguous range of register addresses.

Like registers, FIFO and memory blocks can have names and addresses. The third column always contains a button that displays *Properties* dialog for this FIFO or memory (Figure 4.3).
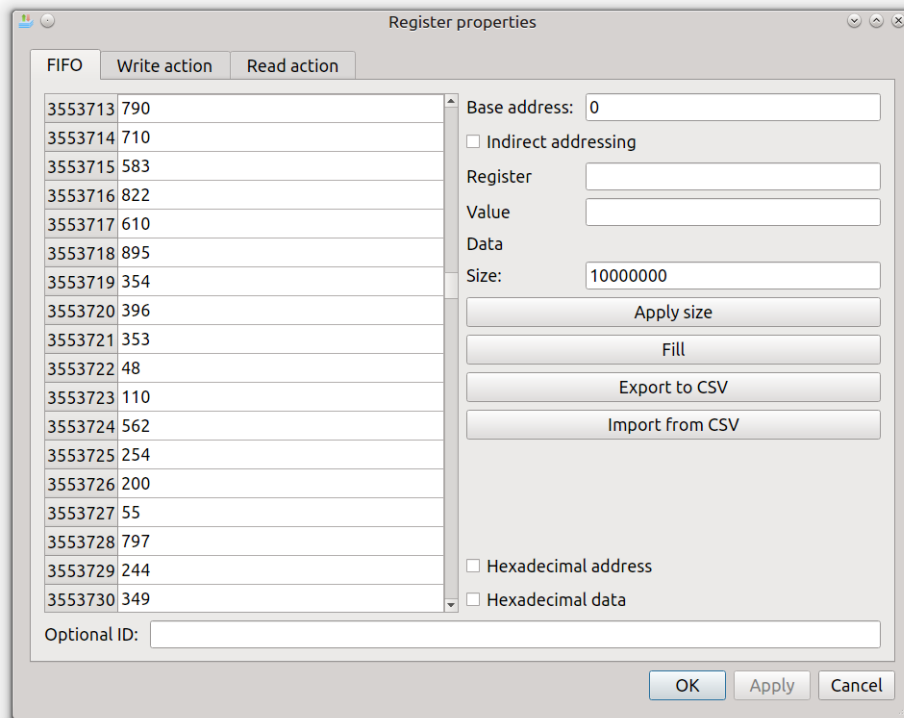
Figure 4.3: *Properties* dialog for *FIFO* type rows

Base address field contains the same value as the second table column. *Indirect addressing* is an optional feature that writes a specified value to the specified register before writing or reading the FIFO or memory.

Data can be exported to a CSV file and imported from a CSV file. Import algorithm is tolerant: it allows any non-digit character or a group of characters to be used as a separator (it doesn't have to be comma, the number of columns also doesn't matter).

Double click on a row number provides a way to scroll the viewport contents to the specified address.

Like registers, FIFO and memory blocks support custom actions (Section 4.3).

4.3 Custom actions

A custom action is a Lua script that is executed whenever the register, FIFO or memory is written and/or read. Custom actions are executed instead of default read/write logic. Write and read actions are separate. Custom actions are set in the *Properties* dialog (Figure 4.4).
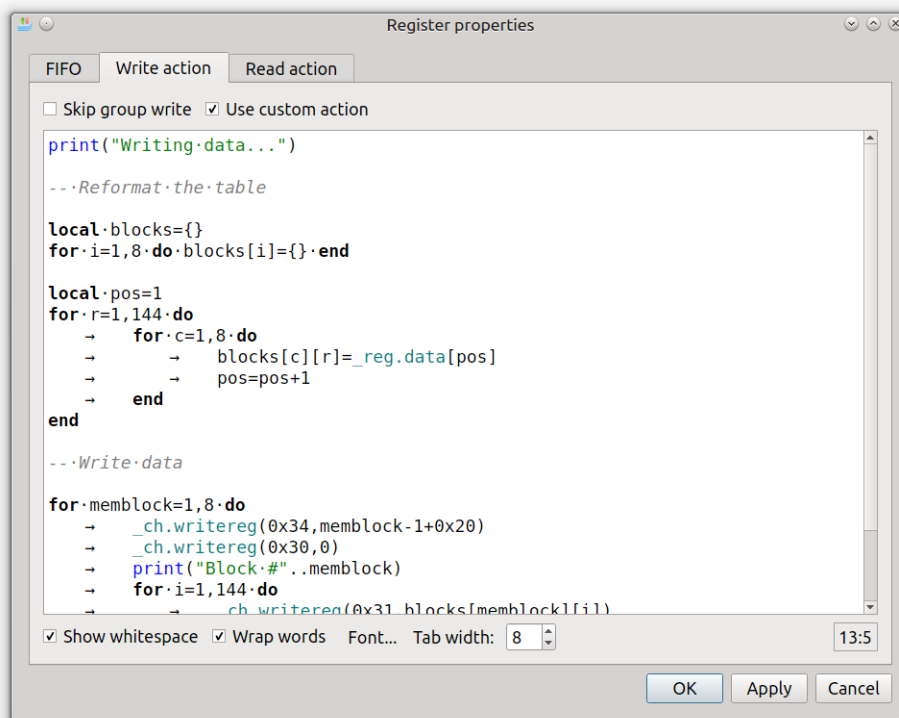
Figure 4.4: Custom write action

When a custom action is triggered, register map executes the attached Lua script. The script is executed as a Lua chunk in the same environment as all the other scripts (including Lua console commands). In addition, the following variables are made available to the custom action handler:

- _ch (*Channel* type object): the channel connected with this register map (Subsection 6.1.5);

- `_map` (*RegisterMap* type object): this register map (Section 6.4);

- `_page` (*integer*): page number;

- `_row` (*integer*): row number;

- `_target` (*string*): `"row"`, `"page"` or `"all"`, depending on what is being written;

- `_reg` (*table*): row data (format is described in Section 6.4).

A write action is not expected to return a value (if it does, the value is ignored). A read action is expected to return the value that is supposed to be displayed (if it doesn't return anything, nothing is changed in the window).

Chapter 5

Plotter window

Plotter windows are used by `sdmconsole` to display data streams. A plotter window can be displayed by clicking a *Stream viewer* button on the sidebar after selecting a source in the object tree. One source can have as many plotter windows as needed. Alternatively, plotter window can be created from a Lua script (see Subsection 6.2.6).

A plotter window has a top toolbar, a scrollable viewport and an optional bottom toolbar (for some view modes).

Plotter window is merely a representation tool; it doesn't provide means to manage the data source itself.

5.1 View modes

Plotter window supports four view modes: *Bars* (for bar charts), *Plot* (for regular line charts), *Bitmap* (for images), *Video* (for video data) and *Binary* (for binary data).

5.1.1 *Bars* and *Plot* modes

Bars and *Plot* modes are used to render bar charts and regular line charts, respectively. Sample numbers, counted from 0, are plotted along the horizontal axis, sample values are plotted along the vertical axis. These modes support multiple layers (Figure 5.1, Figure 5.2). The *Plot* mode provides a bottom toolbar which allows the user to change the line width and toggle antialiasing on/off.
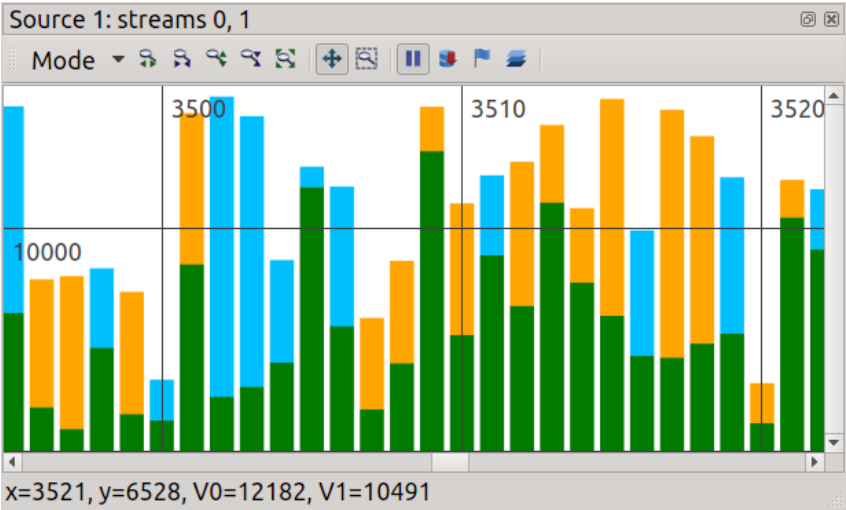
Figure 5.1: Plotter window in *Bars* mode
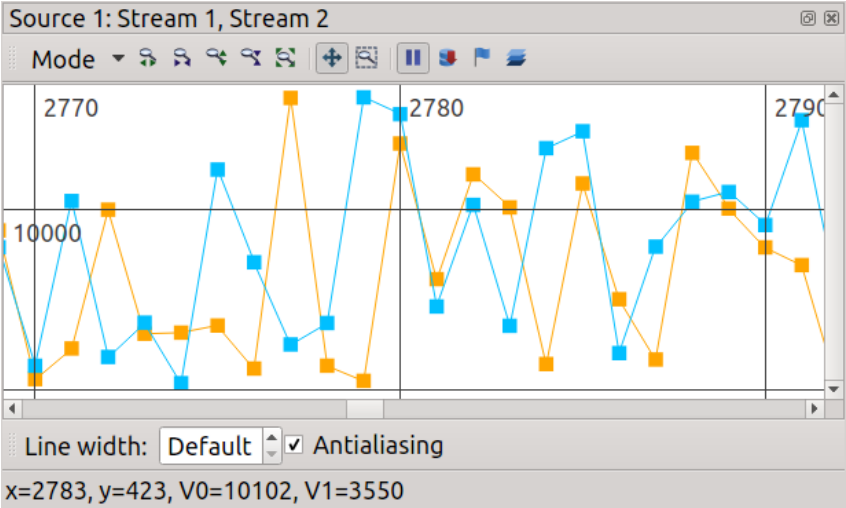
Figure 5.2: Plotter window in *Plot* mode

5.1.2 *Bitmap* mode

Bitmap mode is used to display multiple lines of graphical data (Figure 5.3). Each sample is represented by a pixel. Sample numbers, counted from 0, are plotted along the horizontal axis, packet numbers are plotted along the vertical axis. Sample value is represented by pixel brightness.

In *Bitmap* mode plotter window can display only one layer.
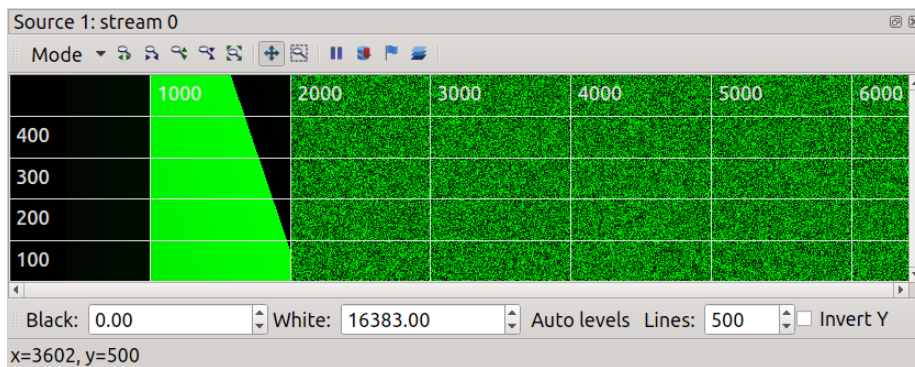
Figure 5.3: Plotter window in *Bitmap* mode

With the bottom toolbar, the user can adjust black and white levels (i.e. mapping between sample values and pixel brightness). If black level is greater than white level, the image is inverted. *Auto levels* button can be used to detect appropriate levels automatically. Number of displayed lines and a place where new lines are added can also be configured.

5.1.3 *Video* mode

Video mode is used to display two-dimensional video data (Figure 5.4). Each sample is represented by a pixel. This mode displays only one packet at a time, line width is set by the user. In this mode plotter window can display only one layer.

The bottom toolbar can be used to adjust black and white levels (similarly to the *Bitmap* mode) and configure the line width.

5.1.4 *Binary* mode

Binary mode is similar to *Bitmap* mode, the difference being that in *Binary* mode a pixel represents one bit instead of sample. Number of bits per sample and bit order can be configured (Figure 5.5).
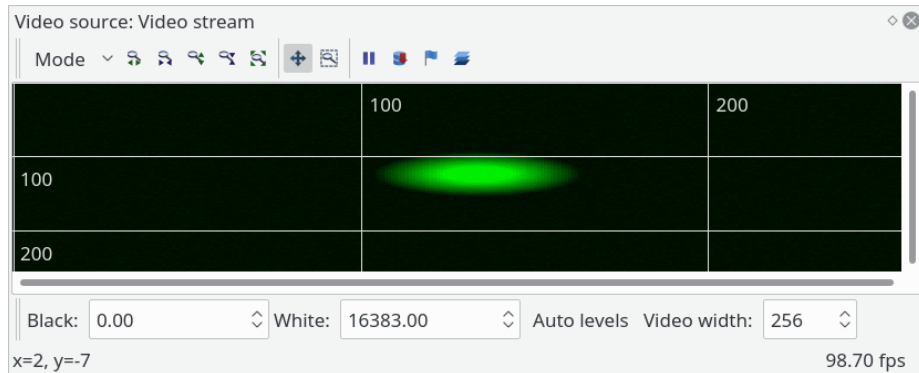
Figure 5.4: Plotter window in *Video* mode
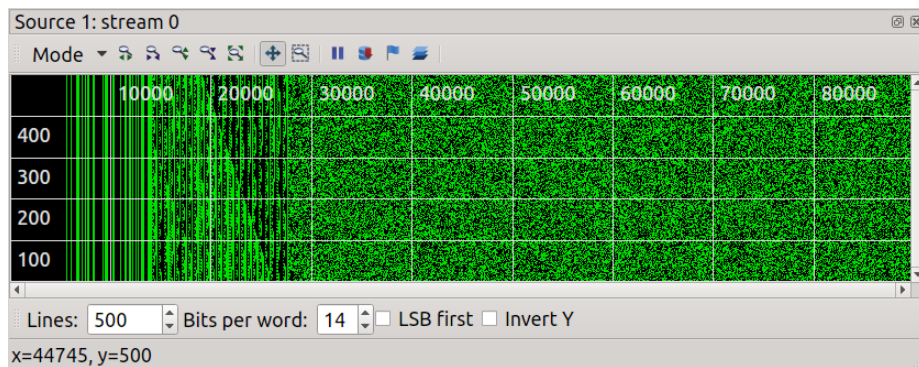
Figure 5.5: Plotter window in *Binary* mode

5.2   Navigation

Plotter window provides multiple ways to navigate the viewport area, which are summarized in Table 5.1. To switch between *Drag to scroll* and *Drag to zoom* modes, press right mouse button while in the viewport area, or use the respective toolbar buttons.

5.3   Managing layers

Layers representation can be configured with the *Layers* dialog which can be accessed using the toolbar button (Figure 5.6). Layer visibility is controlled by a checkbox near the layer name. Colors can be configured on a per-layer basis. Each layer can have a linear transformation applied along the vertical axis; layers are transformed independently of each other. The transformation is based on the following formula:

Table 5.1: Plotter window navigation

Control	To scroll	To zoom
Keyboard	*Up*, *Down*, *Left*, *Right*, *PageUp*, *PageDown* keys	—
Wheel	Scroll up/down, with *Shift*: left/right	With *Control*: zoom vertically, with *Shift+Control*: zoom horizontally
Horizontal wheel	Scroll left/right, with *Shift*: up/down	With *Control*: zoom horizontally, with *Shift+Control*: zoom vertically
Drag and drop	In *Drag to scroll* mode	In *Drag to zoom* mode
Toolbar	—	*Horizontal zoom in*, *Horizontal zoom out*, *Vertical zoom in*, *Vertical zoom out*, *Fit*
Double click	—	Double-click in the viewport area to fit the whole plot (equivalent to the *Fit* toolbar button)

$$y = scale \cdot (x + inputOffset) + outputOffset$$

Most of these features are not available for single-layered modes (*Bitmap*, *Video* and *Binary*); in these modes the user can only choose the currently visible layer (making a new layer visible hides the previously visible layer).

5.4 Examining data

Plotter window status bar displays coordinates and sample values currently hovered by the mouse pointer. The *Add cursor* toolbar button adds a cursor that shows sample values for a specified horizontal position (Figure 5.7). Any number of cursors can be added; existing cursors can be moved by dragging or by using a spinbox. Both status bar and cursors display original sample values, before any layer transformation is applied. Sample values for hidden layers are not displayed.
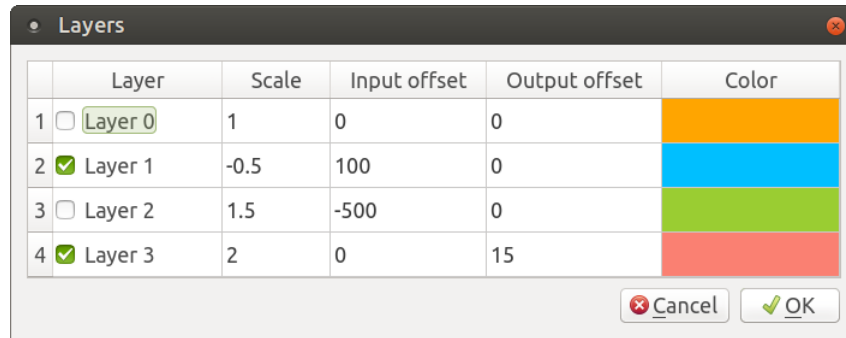
Figure 5.6: Layers configuration dialog
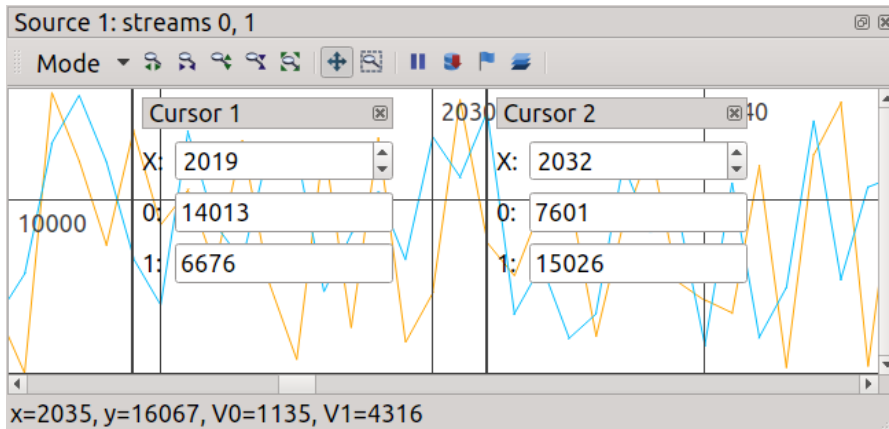
Figure 5.7: Cursors

5.5 Other features

The *Freeze* button prevents plot data from updating while allowing the user to navigate the plot as usual. The *Save image* button exports the contents of the viewport area as an image; both raster (PNG, TIFF, JPEG, BMP) and vector (SVG, PDF) formats are supported. The *Export data* button exports the currently displayed data in the CSV format.

Chapter 6

Scripting engine

SDM integrates a scripting engine based on Lua programming language[2]. The scripting engine uses Lua extension mechanism to provide bindings for the SDM plugin interface as well as (with `sdmconsole`) means to create dialog windows to interact with the user.

The Lua project provides a comprehensive set of documentation which is also shipped with the SDM distribution. The language itself will not be covered in this manual.

Lua extensions provided by the SDM framework are organized as a set of separate libraries, summarized in table 6.1.

Table 6.1: Core Lua extensions provided by SDM

Library	Description
sdm	Provides access to SDM plugins, devices, channels and sources
gui	Provides means to interact with the user via dialog windows (only for `sdmconsole`)
codec	Provides facilities to convert text encoding between UTF-8 (used by SDM) and system encoding

Libraries provided by the SDM framework are object-oriented. Each library provides a global object with the same name which is used to access the library facilities, including creating other objects. Unlike Lua standard libraries, methods must be invoked with a dot, not a colon (i.e. `channel.writereg()`, not `channel:writereg()`). Each object has an *object type* which is a string stored in its `type` field (not method).

---

[2]`http://www.lua.org/`

33

In addition to that, SDM can use Lua extension modules. Modules shipped with the SDM distribution are listed in Table 6.2. More extensions can be added, see Section 6.8. Extension modules are not loaded automatically; to load an extension, use the **require**() Lua function.

Table 6.2: Addon libraries shipped with SDM

Library	Description
luadfic	Invoke functions from external shared libraries
luaipsockets	IP version 4 connectivity
luart	Serial port library

luadfic, luaipsockets and luart are parts of the SDM project, but can be also used with a standalone Lua interpreter.

6.1   sdm library

sdm library provides access to devices through the SDM plugin interface. Its features are available through the sdm global object of *Bridge* type. A simple example:

```lua
-- Open plugin
plugin=sdm.openplugin("testplugin")

-- Open device
device=plugin.opendevice(0)

-- Connect
device.setproperty("IPaddress","127.0.0.1")
device.setproperty("Port","1234")
device.connect()

-- Open channel
channel=device.openchannel(0)

-- Write and read some registers
channel.writereg(0x10,5678)
r=channel.readreg(0x10)
print("Read value: "..r)

-- Open source
source=device.opensource(0)
```

```
-- Read stream data
source.selectreadstreams({0,1})
for i=1,10 do
    data0=source.readstream(0,10000)
    print("0: packet of length "..#data0)
    data1=source.readstream(1,10000)
    print("1: packet of length "..#data1)
    source.readnextpacket()
end
```

6.1.1 `sdm` global object

`sdm.openplugin(filename)`

Opens a plugin file.

Parameters:

    filename (*string*): a plugin file name

Return value:

    Returns a *Plugin* type object.

Remarks:

    If `filename` is not an absolute path, it will be resolved relative to the standard SDM plugins directory (Section 2.5). If `filename` has no extension, a platform-specific extension will be appended (`.dll` for Windows, `.so` for Linux).

    If a plugin can't be opened for any reason, the function will raise an error.

`sdm.plugins([index])`

Queries the number of opened plugins or gets a *Plugin* type object corresponding to an opened plugin.

Parameters:

    index (*integer*, optional): index of an opened plugin (counted from 1)

Return value:

    When called without parameters, returns the number of opened plugins. When called with an `index` parameter, returns a *Plugin* type object

corresponding to an opened plugin with this index.

sdm.info(key)

Queries SDM framework information.

Parameters:

key (*string*): type of information requested

- "host" (*string*): name of the host program ("sdmconsole" or "sdmhost")

- "version" (*string*): SDM framework version ("1.0.3")

- "os" (*string*): operating system type ("windows" or "unix")

- "uilanguages" (*table*): list of user interface languages, where each language is represented by a BCP47 language tag[1] (for example, {"en-US","ru-RU"})

Return value:

Returns the requested value.

sdm.path(name)

Queries an SDM system path.

Parameters:

name (*string*): requested path (see the Remarks section below)

Return value:

Returns an absolute path in a UTF-8 encoded form.

Remarks:

name can be one of the following:

- "currentdir": current directory;

- "homedir": home directory of the current user;

- "currentscript": a path to the currently executed script, or **nil** if it cannot be obtained;

- "currentscriptdir": a path to the directory containing the currently executed script, or **nil** if it cannot be obtained;

---

[1]https://tools.ietf.org/rfc/rfc5646.txt

- "`program`": a path to the host program executable (e.g. `sdmhost` or `sdmconsole`);

- "`installprefix`": SDM installation prefix;

- "`bindir`": SDM binaries directory;

- "`pluginsdir`": SDM plugins directory;

- "`luamodulesdir`": SDM native Lua modules directory;

- "`luacmodulesdir`": SDM binary ("C") Lua modules directory;

- "`qtdir`": Qt plugins directory (only makes sense on Microsoft Windows);

- "`docdir`": SDM documentation directory;

- "`translationsdir`": `sdmconsole` translations directory;

- "`scriptsdir`": SDM scripts directory;

- "`datadir`": SDM plugin data directory.

All directory names are encoded as UTF-8 strings. In order to access files in these directories, portable scripts should use `codec.open()` instead of **`io.open`**`()` since the latter is not Unicode aware. Alternatively, a string can be converted to the local encoding from UTF8 with `codec.utf8tolocal()`, unless the source string contains characters that can't be represented in the local encoding.

sdm.sleep(msec)

Puts the Lua interpreter thread to sleep for a specified time.

Parameters:

    msec (*integer*): time to sleep for, in milliseconds

sdm.time()

Queries the current time.

Return value:

    Returns the amount of time from the start of the epoch, in milliseconds. The meaning of epoch depends on an implementation. These function can be used to measure time intervals.

sdm.lock(action)

Locks or unlocks the SDM plugin interface.

Parameters:

action (*boolean*): **true** to lock, **false** to unlock

Remarks:

When the Lua interpreter invokes an SDM function, sdmconsole will prevent access to the SDM plugin interface from other threads (including the GUI thread). However, other threads still can call SDM API functions while Lua interpreter is executing some other code. Sometimes it can be undesirable, notably when reading stream data. This function obtains exclusive access to the SDM plugin interface.

sdm.lock(**true**) can be called multiple times. In order to release lock, sdm.lock(**false**) must be called the same number of times. If Lua interpreter finishes while the SDM plugin interface is still locked, it will be unlocked automatically.

sdm.lock() is ignored by sdmhost since it doesn't use multiple threads.

sdm.selected()

Gets an item currently selected in the object tree (sdmconsole only).

Return value:

Returns a table with the following fields:

- plugin (*Plugin* type object): currently selected plugin

- device (*Device* type object): currently selected device

- channel (*Channel* type object): currently selected channel

- source (*Source* type object): currently selected source

If a child object is selected, its parent object is also returned as a part of the table (e.g. if *Channel* is selected, *Device* and *Plugin* are also returned). If neither object of the certain type nor its children are selected, the corresponding table field is omitted (which can be tested by comparing its value to **nil**).

6.1.2 Common property interface

sdm library provides a common property interface to access properties of *Plugin*, *Device, Channel* and *Source* type objects.

object.`getproperty(name [, defaultvalue])`

Gets a property value.

Parameters:

    name (*string*): property name
    defaultvalue (*string,* optional): default property value

Return value:

    If the requested property exists, returns its value. If the requested property doesn't exist, returns defaultvalue if it is present, otherwise returns **nil**.

object.`setproperty(name, value)`

Sets the value of name property to value.

Parameters:

    name (*string*): property name
    value (*string*): property value

object.`listproperties(name)`

Gets a property value as a list.

Parameters:

    name (*string*): property name

Return value:

    Returns a table containing list elements.

Remarks:

    Property lists are described in Section 7.4.

sdm library objects also override __index and __newindex metamethods, allowing the user to access properties as ordinary table fields:

```
-- The following statements are equivalent
str=device.getproperty("IPaddress")
str=device.IPaddress
str=device["IPaddress"]

-- The following statements are also equivalent
device.setproperty("IPaddress","127.0.0.1")
device.IPaddress="127.0.0.1"
device["IPaddress"]="127.0.0.1"
```

6.1.3 *Plugin* type object

plugin.**path()**

Gets the path to the plugin file.

Return value:

   Returns a path to the plugin file (absolute, if available).

plugin.**close()**

Closes the plugin.

plugin.**opendevice(id)**

Opens a device.

Parameters:

   id (*integer*): device id (counted from 0)

Return value:

   Returns a *Device* type object.

plugin.**devices([index])**

Queries the number of opened devices for the plugin or gets a *Device* type object corresponding to an opened device.

Parameters:

   index (*integer*, optional): index of an opened device (counted from 1)

Return value:

When called without parameters, returns the number of opened devices for the plugin. When called with an `index` parameter, returns a *Device* type object corresponding to an opened plugin with this index.

Remarks:

Note that `index` denotes a position of the device in the object tree and is not the same as device id.

plugin.**getproperty(name [, defaultvalue])**
plugin.**setproperty(name, value)**
plugin.**listproperties(name)**

See Subsection 6.1.2.

6.1.4 *Device* type object

device.**id()**

Gets the device id.

Return value:

Returns the id of the device.

device.**close()**

Closes the device.

device.**connect()**

Connects to the device.

Remarks:

If the device requires connection parameters, they are set using the property interface (see Subsection 6.1.2).

device.**disconnect()**

Disconnects from the device.

device.**isconnected()**

Queries the device connection status.

Return value:

Returns **true** for an active connection, **false** otherwise.

device.**openchannel(id)**

Opens a channel.

Parameters:

id (*integer*): channel id

Return value:

Returns a *Channel* type object.

device.**channels([index])**

Queries the number of opened channels for the device or gets a *Channel* type object corresponding to an opened channel.

Parameters:

index (*integer*, optional): index of an opened channel (counted from 1)

Return value:

When called without parameters, returns the number of opened channels for the device. When called with an index parameter, returns a *Channel* type object corresponding to an opened channel with this index.

Remarks:

Note that index denotes a position of the channel in the object tree and is not the same as channel id.

device.**opensource(id)**

Opens a source.

Parameters:

id (*integer*): source id

Return value:

Returns a *Source* type object.

device.**sources([index])**

Queries the number of opened sources for the device or gets a *Source* type object corresponding to an opened source.

Parameters:

    index (*integer*, optional): index of an opened source (counted from 1)

Return value:

    When called without parameters, returns the number of opened sources for the device. When called with an index parameter, returns a *Source* type object corresponding to an opened source with this index.

Remarks:

    Note that index denotes a position of the source in the object tree and is not the same as source id.

device.**getproperty(name [, defaultvalue])**
device.**setproperty(name, value)**
device.**listproperties(name)**

See Subsection 6.1.2.

6.1.5 *Channel* type object

channel.**id()**

Gets the channel id.

Return value:

    Returns the id of the channel.

channel.**close()**

Closes the channel.

channel.**writereg(addr, value)**

Writes to a register.

Parameters:

    addr (*integer*): register address
    value (*integer*): value to write

channel.**readreg(addr)**

Reads from a register.

Parameters:

addr (*integer*): register address

Return value:

Returns a value read from the register.

channel.writefifo(addr, data [, flags])

Writes data to the FIFO with address addr.

Parameters:

addr (*integer*): register address
data (*table*): data to write (table of integers)
flags (*string*, optional): a comma-separated list of flags:

- "all": blocking operation, disallow partial write
- "part": blocking operation, allow partial write
- "nb": non-blocking operation
- "start": this operation starts a new packet

Only one of "all", "part" and "nb" can be specified. Default is "all". See also the Remarks below.

Return value:

Returns the number of words written to the FIFO. In the non-blocking mode, returns **nil** if nothing was written because write operation would block.

Remarks:

FIFO represents a memory block mapped to a single register address. In the blocking mode, if partial write is not allowed, the function returns when (1) all the data is written or (2) an error occurs. If partial write is allowed, in addition, the function can return when at least one word is written and no more data can be written without blocking. In the non-blocking mode, in addition, the function returns when no data can be written without blocking.

See Section 9.3 for detailed description of SDM FIFO semantics.

channel.readfifo(addr, n [, flags])

Reads n words from the FIFO with address addr.

Parameters:

addr (*integer*): register address

n (*integer*): number of words to read

flags (*string*, optional): a comma-separated list of flags:

- "all": blocking operation, disallow partial read
- "part": blocking operation, allow partial read
- "nb": non-blocking operation
- "next": proceed to the next packet

Only one of "all", "part" and "nb" can be specified. Default is "all". See also the Remarks below.

Return value:

Returns the number of words read from the FIFO. If the FIFO supports the notion of packets, returns 0 at the end of packet. In the non-blocking mode, returns **nil** if nothing was read because the operation would block.

Remarks:

FIFO represents a memory block mapped to a single register address. In the blocking mode, if partial read is not allowed, the function returns when (1) the requested number of words is read, (2) packet ends or (3) an error occurs. If partial read is allowed, in addition, the function can return when at least one word is read and no more data are available for immediate reading. In the non-blocking mode, in addition, the function returns when no data can be read without blocking.

To proceed to the next packet, the function must be called with the "next" flag. See Section 9.3 for detailed description of SDM FIFO semantics.

channel.writemem(addr, data)

Writes data to a memory block starting at addr.

Parameters:

addr (*integer*): register address

data (*table*): data to write

Remarks:

A memory block is a block of consecutive registers. See Section 9.3 for detailed description of SDM memory block semantics.

channel.readmem(addr, n)

Reads n words from a memory block starting at addr.

Parameters:

   addr (*integer*): register address

   n (*integer*): number of words to read

Return value:

   Returns a *table* read from the requested address.

Remarks:

   A memory block is a block of consecutive registers. See also Section 9.3
for detailed description of SDM memory block semantics.

channel.registermap()

Opens a register map for the channel (sdmconsole only).

Return value:

   Returns a *RegisterMap* type object.

Remarks:

   If a register map windows for this channel is not opened, this function
opens it. Otherwise it raises the existing window.

channel.getproperty(name [, defaultvalue])
channel.setproperty(name, value)
channel.listproperties(name)

See Subsection 6.1.2.

6.1.6   *Source* type object

source.id()

Gets the source id.

Return value:

   Returns the id of the source.

source.close()

Closes the source.

source.selectreadstreams(streams [, packets [, df]])

Selects data streams to read.

Parameters:

    streams (*table*): stream ids to select (table of integers)

    packets (*integer,* optional): suggested minimum number of packets to deliver consecutively (per stream), the value of 0 leaves the decision to the plugin, default value is 0

    df (*integer,* optional): suggested decimation factor (a non-binding request to reduce the source bandwidth by the factor of df), 1 or greater, default value is 1

Remarks:

    Note: calling this functions resets the error counter.
    See also Section 9.4 for description of SDM stream semantics.

source.readstream(stream, n [, flag])

Reads n samples from a stream with id stream.

Parameters:

    stream (*integer*): stream id
    n (*integer*): number of samples to read
    flag (*string,* optional): one of the following:

        – "all": blocking operation, disallow partial read

        – "part": blocking operation, allow partial read

        – "nb": non-blocking operation

    Default is "all". See also the Remarks below.

Return value:

Returns a table of samples read from the stream or empty table at the end of packet. In the non-blocking mode, in addition, returns **nil** if nothing was read because an operation would block.

Remarks:

The requested stream must be selected with selectreadstreams().

Data from all streams that were received simultaneously will be delivered also simultaneously. To proceed to the next packet, readnextpacket() must be called (see below).

In the blocking mode, if partial read is not allowed, the function returns when (1) the requested number of samples is read, (2) packet ends or (3) an error occurs. If partial read is allowed, in addition, it returns when at least one sample is read and no more data are available for immediate

reading. In the non-blocking mode, in addition, the function returns when no data are available for immediate reading.

See also Section 9.4 for description of SDM stream semantics.

source.readnextpacket()

Proceeds to the next packet. This function affects all selected streams.

source.discardpackets()

Discards data from the receiver buffer. If the stream supports the notion of packets, the next operation will read from the start of the packet.

Remarks:

Note: calling this functions resets the error counter.

source.readstreamerrors()

Return value:

Returns the number of stream errors for the source.

Remarks:

Stream error is a condition when consecutive operations read non-consecutive data.

source.addviewer(streams [, mode [, multilayer]])

Opens a plotter window to display stream data for the source (sdmconsole only).

Parameters:

streams (*table*): a set of stream ids to display

mode (*string*): viewer mode: "bars", "plot", "bitmap", "video" or "binary"

multilayer (*integer*): a non-zero value activates the *Cycle through layers* option and sets the number of layers to cycle through (ignored if streams contains more than 1 element)

source.getproperty(name [, defaultvalue])
source.setproperty(name, value)
source.listproperties(name)

See Subsection 6.1.2.

6.2   `gui` library

gui library is an `sdmconsole` Lua extension allowing the script developer to create dialog boxes to interact with the user. It includes simple modal dialogs (such as message boxes) that are created by calling methods of the gui global object (of *DialogServer* type) and destroyed before the method returns, as well as more complex modeless windows that are represented as separate objects.

6.2.1   `gui` global object

`gui.screen()`

Gets the dimensions of the active screen.

Return value:

Returns the application's active screen dimensions in pixels (width and height, as 2 return values).

`gui.messagebox(message [, title [, icon [, buttons]]])`

Displays a modal message box (Figure 6.1).
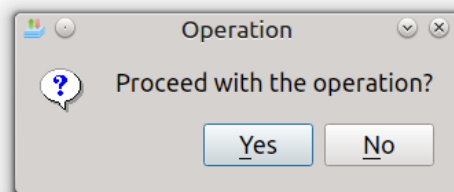
Figure 6.1: Message box window

Parameters:

message (*string*): a message to display
title (*string*, optional): window title
icon (*string*, optional): icon, one of the following:

  – `"error"`
  – `"warning"`
  – `"information"`

- "question"
- "" (no icon)

buttons (*string*, optional): buttons, a comma-separated string contain-
ing one or more of the following: "ok", "cancel", "yes", "no"; the
first button in the string will be selected by default.

Return value:

Returns a string corresponding to pressed button.

Examples:

```
r=gui.messagebox("Proceed with the
    operation?","Operation","question","yes,no")
if r~="yes" then return end
```

gui.inputdialog(title, label [, value])

Displays a simple modal dialog box allowing the user to enter a string
(Figure 6.2) or select an option from a drop-down list (Figure 6.3).
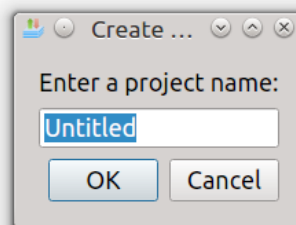
Figure 6.2: Input dialog window (with a text input field)

Parameters:

title (*string*): window title

label (*string*): label (text that is displayed in the input dialog box)

value (*string* or *table*, optional): a string containing initial value of the
text input field, or a table consisting of strings to populate the drop-down
list
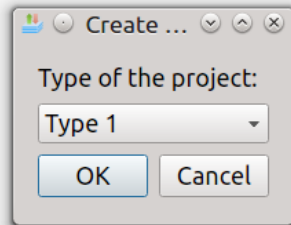
Figure 6.3: Input dialog window (with a drop-down list)

Return value:

Returns a string entered by the user or selected from the drop-down list, or **nil** if *Cancel* button was pressed.

Remarks:

When `value` is not present, the user is asked to enter a string in a text input field. When `value` is of *string* type, the field is initialized with the provided string. When `value` is of *table* type, the text input field is replaced with a drop-down list containing strings from the table.

For more complex input, consider using a *FormDialog* type object (Subsection 6.2.4).

Examples:

```
projectname=gui.inputdialog("Create project","Enter a project
    name:","Untitled")
projecttype=gui.inputdialog("Create project","Type of the
    project:",{"Type 1","Type 2"})
```

gui.filedialog(mode [, filter])

Displays a standard modal dialog box allowing the user to choose a file or a directory (Figure 6.4).

Parameters:

   mode (*string*): `"open"`, `"save"` or `"dir"`
   filter (*string*, optional): file name filter
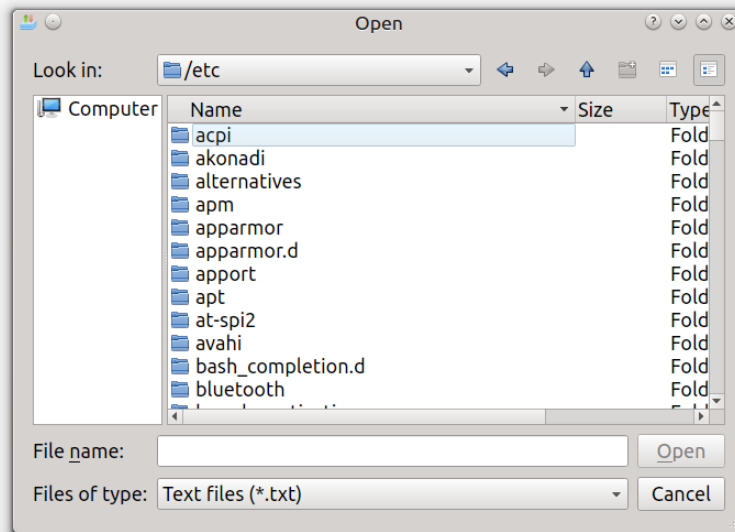
Figure 6.4: File dialog window

Return value:

Returns a path to the selected file, or **nil** if *Cancel* was pressed.

Remarks:

A file name filter is a string containing a file name mask and an optional description. Multiple filters can be separated by a double semicolon. For example:

```
"*.txt"
"Text files (*.txt)"
"Image files (*.png *.jpg *.gif *.bmp)"
"Text files (*.txt);;All files (*)"
```

"*.*" should not be used as a mask for all files since some files don't have an extension.

Examples:

```
filename=gui.filedialog("open","Text files (*.txt);;All files
    (*)")
if not filename then return end
```

gui.createdialog(dialog [, ...])

Creates a modeless dialog window.

Parameters:

dialog (*string*): dialog type, one of the following:

- "progress" (a progress bar window)
- "form" (a table-based form window)
- "textviewer" (a text viewer window)
- "plotter" (a plotter window)

Other parameters, if any, are passed to the dialog constructor.

Return value:

Returns a dialog object. Type of the object depends on dialog argument.

Remarks:

The dialog window is hidden after creation; to show it, invoke the show() method with **true** as an argument (the plotter window is an exception to this). Similarly, the dialog object is not destroyed when the dialog is closed; this allows the developer to query it for information (e.g. user input). To destroy the dialog object, invoke the close() method. Alternatively, the dialog can be destroyed automatically by the owner.

The dialog object's lifecycle is managed by the Lua server. When the object goes out of scope in Lua, it becomes eligible for destruction by the Lua garbage collector. The dialog will also be destroyed when the application main window is closed.

6.2.2   Common modeless dialog interface

gui library provides a common interface to manage modeless windows represented by *ProgressDialog*, *FormDialog*, *TextViewer* and *Plotter* type objects.

***dialog*.show([action])**

Queries the window visibility state and optionally shows or hides the window.

Parameters:

action (*boolean*, optional): **true** to show, **false** to hide

Return value:

Returns previous window visibility state.

***dialog*.close()**

Closes the window and destroys the object, freeing all the resources.

Remarks:

Attempts to access the dialog's methods after it has been destroyed will result in an error being raised.

***dialog*.settitle([title])**

Queries the current window title and optionally sets it to a new value.

Parameters:

title (*string*, optional): new window title

Return value:

Returns previous window title.

***dialog*.resize([width, height])**

Queries the current window size and optionally resizes the window.

Parameters:

width (*integer*, optional): new width
height (*integer*, optional): new height

Return value:

Returns previous window width and height as 2 return values.

Remarks:

width and height must be either both present or both absent.

***dialog*.move()**
***dialog*.move(left, top)**

***dialog*.move("center")**

Queries the current window position and optionally moves the window.

Parameters:

　left (*integer*, optional): new x coordinate of the top-left corner
　top (*integer*, optional): new y coordinate of the top-left corner
　"center" (*string*, optional)

Return value:

　Returns previous coordinates of the top-left corner as 2 return values.

Remarks:

　When invoked in the first form, queries current top-level corner position. In the second form moves the window to the specified position on the desktop. In the third form centers the window on the desktop.

6.2.3　*ProgressDialog* type object

A progress dialog is used to display the status of a long operation. It can be created by the gui.createdialog() function. An example is shown on Figure 6.5.

```
progress=gui.createdialog("progress","Waiting for
    completion...",0,10)
progress.setvalue(0)
for i=1,10 do
    sdm.sleep(1000)
    progress.setvalue(i)
    if progress.canceled() then
        progress.close()
        return
    end
end
```

gui.createdialog("progress")
gui.createdialog("progress", text)
gui.createdialog("progress", text, max)
gui.createdialog("progress", text, min, max)

Creates a progress dialog.
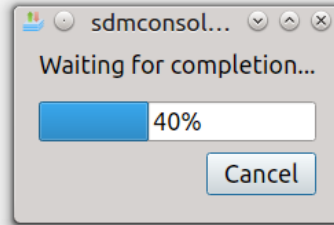
Figure 6.5: Progress dialog window

Parameters:

    `text` (*string*, optional): text displayed in the dialog box

    `min` (*integer*, optional): minimum value (`0` if omitted)

    `max` (*integer*, optional): maximum value (`100` if omitted)

Return value:

    Returns a *ProgressDialog* type object.

progress`.setvalue(value)`

Sets a progress bar value.

Parameters:

    `value` (*integer*): new progress bar value

Remarks:

    The value must be in the range specified by the `setrange()` method or the constructor. Invoking `setvalue(0)` shows the dialog automatically; it is not needed to call `show(`**`true`**`)`. Invoking `setvalue()` with a maximum value as an argument hides the dialog.

progress`.settext(text)`

Sets a text string to be displayed in the dialog box.

Parameters:

    `text` (*string*): new text string to display

progress`.setrange(min, max)`

Sets the progress bar range.

Parameters:

    min (*integer*): minimum value

    max (*integer*): maximum value

Remarks:

Sometimes the maximum value is not known (i.e. when downloading a file of unknown size). In this case both min and max should be set to 0. The dialog must be shown explicitly with show(**true**); do not call setvalue(0) since it will close the dialog.

progress.reset()

Resets the progress bar to an initial state and hides the dialog window.

progress.canceled()

Queries whether the *Cancel* button has been clicked.

Return value:

Returns **true** if the *Cancel* button has been clicked or **false** otherwise. After the *Cancel* button has been clicked, this method will return **true** until reset() is invoked.

progress.show([action])
progress.close()
progress.settitle([title])
progress.resize([width, height])
progress.move(...)

See Subsection 6.2.2.

6.2.4 *FormDialog* type object

A form dialog is used to request more complex user input than is possible by using gui.inputdialog(). It can be created by the gui.createdialog() function. An example is shown on Figure 6.6.

```
form=gui.createdialog("form","Enter user data:")
form.addtextoption("Name","John Smith")
form.addlistoption("Gender",{"M","F","Other"})
form.addfileoption("Photo","open","","JPEG Images (*.jpg
    *.jpeg);;All files (*)")
r=form.exec()
if not r then return end
print("Name is "..form.getoption(1))
print("Gender is "..form.getoption(2))
print("Photo file path is "..form.getoption(3))
```
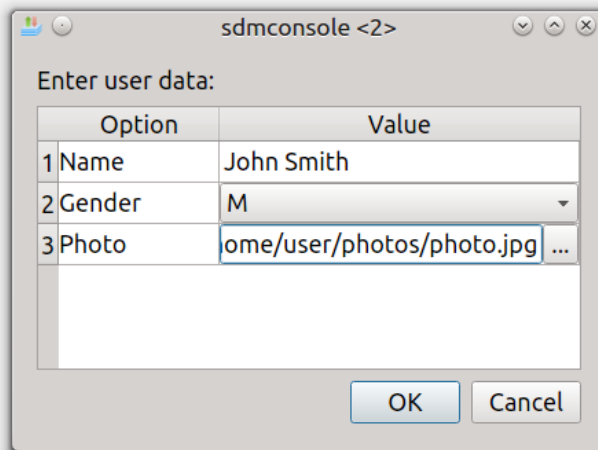

Figure 6.6: Form dialog window

gui.createdialog("form" [, text])

Creates a form dialog.

Parameters:

    text (*string*, optional): text displayed in the dialog box

Return value:

    Returns a *FormDialog* type object.

form.settext([text])

Queries the current text string displayed in the dialog box and optionally sets it to the new value.

Parameters:

　　text (*string*, optional): new text string to display

Return value:

　　Returns a previous dialog text string.

form.addtextoption(name [, value])

Adds a new text input field to the form.

Parameters:

　　name (*string*): option name
　　value (*string*, optional): option value

form.addlistoption(name, options [, default])

Adds a new drop-down list to the form.

Parameters:

　　name (*string*): option name
　　value (*table*): options for the drop-down list
　　default (*number* or *string*, optional): default option

Remarks:

　　default is interpreted as either option index (if it is a number) or option value. If default is omitted, the first option is selected by default.

form.addfileoption(name, mode [, filename [, filter]])

Adds a new file selector to the form.

Parameters:

　　name (*string*): option name
　　mode (*string*): file selector mode ("open", "save" or "dir")
　　filename (*string*, optional): default value
　　filter (*string*, optional): file name filter, as in gui.filedialog()

form.getoption(name_or_index)

Gets an option value chosen by the user.

Parameters:

　　name_or_index (*string* or *integer*): either an option name or its index (counted from 1)

Return value:

    Returns two values:

    1. the string selected by the user,

    2. option index in the drop-down list, counted from 1 (or **nil** if this option doesn't have a drop-down list).

form.exec()

Execute a dialog, waiting until the user either accepts or dismisses it.

Return value:

    Returns **true** if the dialog was accepted or **false** if it was dismissed.

form.show([action])
form.close()
form.settitle([title])
form.resize([width, height])
form.move(...)

See Subsection 6.2.2.

6.2.5   *TextViewer* type object

Displays text data. *TextViewer* can be created by the `gui.createdialog()` function. An example is shown on Figure 6.7.

    It is recommended that the contents are initialized before the window is shown to allow the dialog to automatically set a suitable window size.

```
tv=gui.createdialog("textviewer")
tv.setfont("monospace",12)
tv.setfile("lipsum.txt")
tv.show(true)
```

gui.createdialog("textviewer" [, title])

Creates a text viewer dialog.

Parameters:

    `title` (*string*, optional): window title
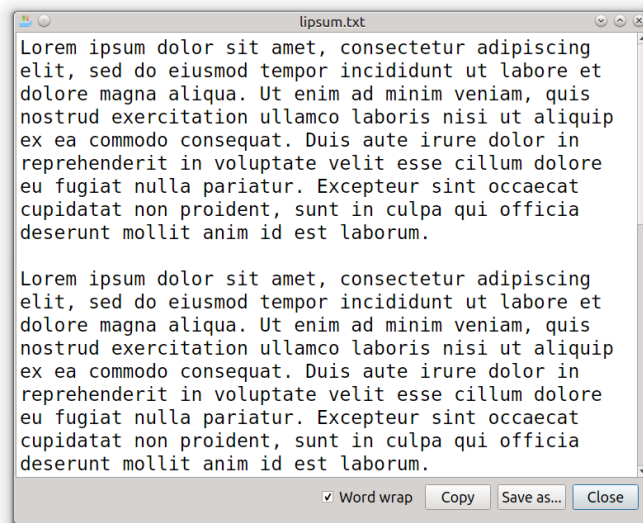
Figure 6.7: Text viewer window

Return value:

Returns a *TextViewer* type object.

textviewer.setfont(name [, size [, subfamily]])

Sets the font used to display text.

Parameters:

name (*string*): font name (see the Remarks section below)
size (*number*, optional): point size
subfamily (*string*, optional): font subfamily ("regular", "italic",
"bold" or "bold italic")

Remarks:

name can be either a font family name, like "Times", "Helvetica" or
"Courier", or one of the special values listed below:

- "default": default user interface font

- "sans-serif": default sans-serif font

- "serif": default serif font

- "monospace": default fixed width font

***textviewer*`.settext(text)`**

Replaces window contents with `text`.

Parameters:

    `text` (*string*): text to display

***textviewer*`.addtext(text)`**

Adds `text` to the window.

Parameters:

    `text` (*string*): text to add

***textviewer*`.setfile(filename)`**

Replaces window contents with the data from `filename`.

Parameters:

    `filename` (*string*): file name

***textviewer*`.setwrap(mode)`**

Sets the word wrap mode.

Parameters:

    `mode` (*boolean*): word wrap mode (true to wrap lines at word boundaries, false otherwise)

***textviewer*`.clear()`**

Clears the window.

***textviewer*`.show([action])`**
***textviewer*`.close()`**
***textviewer*`.settitle([title])`**
***textviewer*`.resize([width, height])`**
***textviewer*`.move(...)`**

See Subsection 6.2.2.

6.2.6 *Plotter* **type object**

A plotter is used to display numerical data in a graphic form. While not technically a dialog, it is also created by the `gui.createdialog()` function

and supports the common modeless dialog interface. An example is shown on Figure 6.8.

```
plotter=gui.createdialog("plotter")
plotter.adddata({1,10,2,9,3,8,4,7,5,6})
```
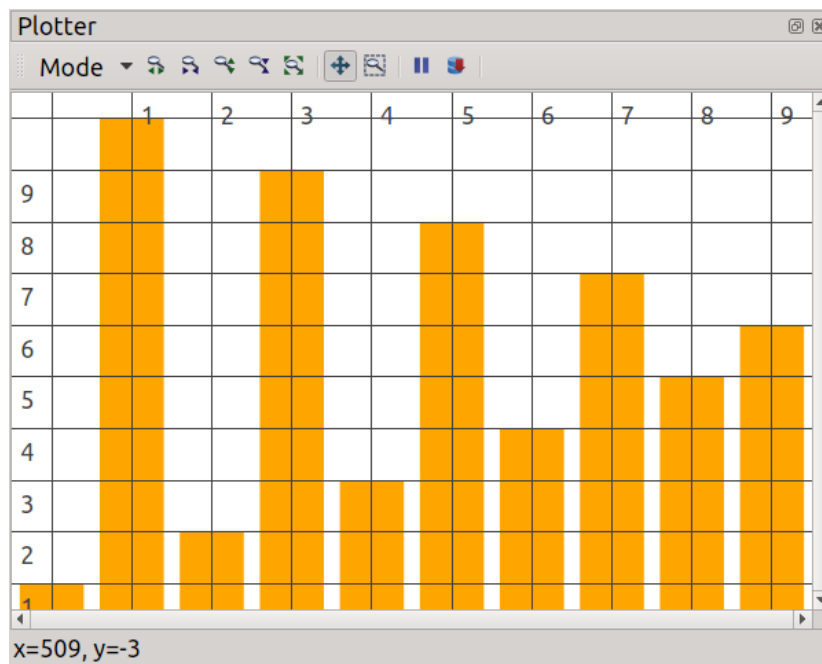

Figure 6.8: Plotter window

gui.createdialog("plotter" [, mode])

Creates a plotter window.

Parameters:

    mode (*string*, optional): data representation mode:

- preferred (either "bars" or "plot", depending on the last user choice; this is the default mode)
- bars (a bar chart)
- plot (an ordinary 2D graph)
- bitmap (a multi-line bitmap)
- video (two-dimensional video data)

– `binary` (multi-line binary data)

Return value:

Returns a *Plotter* type object.

plotter`.adddata(data [, layer])`

Adds a line of data to the plot.

Parameters:

`data` (*table*): data to add
`layer` (*integer*, optional): layer (default is `0`)

Remarks:

In `"bars"` and `"plot"` modes the plotter can draw multiple layers, but only one line at a time (adding a new line erases the previous one). In `"bitmap"` and `"binary"` modes it can draw multiple lines, but only one layer. In `"video"` mode plotter displays a single layer and a single packet of data which is split into multiple lines according to the video width set by the user (or `setoption` method). To create a new layer in the `"bitmap"`, `"video"` or `"binary"` mode, the existing layer must be removed first (see the `removedata()` method).

plotter`.removedata([count [, layer]])`

Removes last `count` lines of data from the plot's `layer`. If `count` equals `-1` or is equal or greater than the number of stored lines, the entire layer is removed.

Parameters:

`count` (*integer*, optional): number of lines to remove (default is `-1`, that is, remove the entire layer)
`layer` (*integer*, optional): layer (default is `0`)

plotter`.setmode([mode])`

Queries a current data representation mode and optionally sets it.

Parameters:

`mode` (*string*, optional): new mode (`"preferred"`, `"bars"`, `"plot"`, `"bitmap"`, `"video"` or `"binary"`)

Return value:

Returns the previous mode.

plotter`.setoption(name [, value])`

Queries a plotter option `name` and optionally sets it to `value`.

Parameters:

    name (*string*): option name (see the *Remarks* below)
    value (type depends on `name`, optional): a value to set

Return value:

    Returns the previous option value.

Remarks:

    A list of possible options depends on `mode`.
    `"bars"` and `"plot"` modes define the following option:

- `"linewidth"` (*integer*): line width for the `"plot"` mode

- `"antialiasing"` (*bool*): use antialiasing for the `"plot"` mode

    Note that while these options can be set for both `"bars"` and `"plot"` modes, they are only used by the `"plot"` mode. Line width of 0 (the default value) instructs the plotter to use a cosmetic pen, which is 1 pixel wide when rasterized on the screen, but can be different when exported to vector formats (SVG or PDF).
    `"bitmap"` mode defines the following options:

- `"lines"` (*integer*): number of lines to display

- `"blackpoint"` (*number*): black point value

- `"whitepoint"` (*number*): white point value

- `"inverty"` (*boolean*): invert Y axis

    `"video"` mode defines the following options:

- `"blackpoint"` (*number*): black point value

- `"whitepoint"` (*number*): white point value

- `"videowidth"` (*boolean*): video width

    `"binary"` mode defines the following options:

- `"lines"` (*integer*): number of lines to display

- `"bits"` (*integer*): number of significant bits per sample

- `"lsbfirst"` (*boolean*): put the least significant bit first

- `"inverty"` (*boolean*): invert Y axis

plotter.`setlayeroption(layer, name [, value])`

Queries a layer option `name` and optionally sets it to `value`.

Parameters:

    `layer` (*integer*): layer
    `name` (*string*): option name (see the *Remarks* below)
    `value` (type depends on `name`, optional): a value to set

Return value:

    Returns the previous layer option value.

Remarks:

    All modes support the following layer options:

- `"name"` (*string*): layer display name

- `"visibility"` (*boolean*): layer visibility

    In `"bars"` and `"plot"` modes any number of layers can be visible; in other modes, only one layer can be visible at any time (making another layer visible hides the currently visible layer).

    In addition, `"bars"` and `"plot"` modes support the following layer options:

- `"scale"` (*number*): layer transformation scale

- `"inputoffset"` (*number*): layer transformation input offset

- `"outputoffset"` (*number*): layer transformation output offset

- `"color"` (*string*): layer color name

Layer transformation is based on the following formula:

$$y = scale \cdot (x + inputOffset) + outputOffset$$

Color names can be represented as in HTML or CSS ("#RRGGBB"), SVG named colors are also supported[1].

***plotter*`.zoom([x, y])`**

Changes plotter window display scale.

Parameters:

    x (*number*, optional): x scale ratio
    y (*number*, optional): y scale ratio

Remarks:

    When called without arguments, fits the whole image in the window (equivalent to the *Zoom fit* toolbar button).

    When called with x and y arguments, multiplies the horizontal and vertical scale coefficients by x and y respectively.

***plotter*`.addcursor(pos [, name])`**

Adds a cursor to the plotter window.

Parameters:

    pos (*integer*): cursor position
    name (*string*, optional): cursor name

***plotter*`.show([action])`**
***plotter*`.close()`**
***plotter*`.settitle([title])`**
***plotter*`.resize([width, height])`**
***plotter*`.move(...)`**

See Subsection 6.2.2. Note that window management functions such as `resize()` and `move()` will not work properly when the plotter window is docked.

6.3 codec library

`codec` library handles text encoding conversion. Its features are accessed through the `codec` global object of *Codec* type.

---

[1]`http://www.w3.org/TR/SVG/types.html#ColorKeywords`

SDM internally uses UTF-8 to represent all text. Native UTF-8 support provided by Lua is limited: it can work with UTF-8 strings as byte arrays and convert them to arrays of code points (since Lua 5.3), but for all interactions with the operating system (file names, environment variables, etc.) a locale-dependent multibyte encoding is used. This is not usually a problem for Linux since it likely uses UTF-8 anyway, but Windows uses legacy "ANSI" and "OEM" encodings which can't represent all Unicode characters. On the other hand, Windows provides a separate UTF-16 API, but standard Lua libraries make no use of it.

The functions of the `codec` library are provided to partially mitigate the aforementioned encoding problems. Some of them, like `codec.dofile()` and `codec.open()`, are designed to be replacements for the similar functions from the Lua standard library. The standard functions are not replaced by default; if desired, one can replace them as follows:

```
dofile=codec.dofile
io.open=codec.open
```

6.3.1   **codec** global object

codec.utf8tolocal(str)

Converts a UTF-8 string to a locale-dependent multibyte encoding.

Parameters:

   `str` (*string*): a UTF-8 string

Return value:

   Returns the converted string.

Remarks:

   More advanced text encoding conversion features are provided by the codec FSM (Subsection 6.3.2).

codec.localtoutf8(str)

Converts a string in a locale-dependent multibyte encoding to a UTF-8 string.

Parameters:

str (*string*): a string in local encoding

Return value:

Returns the converted string.

Remarks:

More advanced text encoding conversion features are provided by the codec FSM (Subsection 6.3.2).

codec.print(str)
codec.write(str)

Writes a UTF-8 encoded text string to the standard output. The difference between codec.print() and codec.write() is that the former will append a newline character, while the latter won't.

Parameters:

str (*string*): a UTF-8 string

Remarks:

Standard stream encoding is only converted when the stream is detected to be bound to a console (terminal), otherwise stream encoding is unchanged. This allows the output to be displayed correctly in sdmconsole which expects UTF-8 encoded data.

codec.dofile([filename])

Executes a Lua script from a file with a UTF-8 encoded name. When called without arguments, reads from the standard input.

Parameters:

filename (*string*, optional): file name (UTF-8 encoded)

Return value:

Returns all values returned by the chunk (if any).

Remarks:

This function is similar to the **dofile**() function from the Lua standard library, but is not affected by Windows-related file name encoding problems since it uses wide-character API to open files, making it possible to work with file names containing characters not present in the current ANSI code page.

If filename is not an absolute path, the function will use the following rules to resolve it:

1. If the code currently being executed originates from a file, try to resolve the path relative to the directory containing that file.

2. Failing that, try to resolve the path relative to the SDM Lua modules directory intended for backend scripts (Section 2.5).

3. Failing that, try to resolve the path relative to the SDM scripts directory intended for user-visible scripts (Section 2.5).

4. Failing that, try resolve the path relative to the current directory.

5. Failing that, raise an error.

This is different from the standard **dofile()** behavior which is always to use the current directory.

codec.open(filename [, mode])

Opens a file with a UTF-8 encoded name.

Parameters:

    filename (*string*): file name (UTF-8 encoded)

    mode (*string*, optional): access mode, like in **io.open**(): "r", "w", "a", "r+", "w+" or "a+"; default is "r"; b can be added to the mode string as a second or third character to force binary mode on Microsoft Windows

Return value:

Returns a file handle that is compatible with standard Lua functions from the **io** library, or **nil** and an error message in case of errors.

Remarks:

This function is designed to be a drop-in replacement for the standard **io.open()** function. Unlike the latter, this function assumes the file name to be encoded in UTF-8. Under Microsoft Windows it uses wide-character API to open files, which makes it possible to work with file names containing characters not present in the current ANSI code page.

Only file names are transcoded, contents are in no way altered.

codec.createcodec(encoding)

Creates a state machine for conversion between UTF-8 and the selected encoding.

Parameters:

encoding (*string*): see below

Return value:

Returns a *CodecFSM* type object.

Remarks:

Character encodings recognized by the library are listed in Table 6.3. Some of these encodings can be unsupported on certain platforms. Hyphens and underscores in encoding names are ignored. `"local"` is a locale-dependent multibyte encoding, `"ansi"` is a synonym for `"local"`, `"oem"` is an OEM encoding on Microsoft Windows and equivalent to `"local"` on other platforms. `"utf-16"` endianness is native to the platform. `"wchar_t"` is a native wide character encoding equivalent to `"utf-16"` on Microsoft Windows. Byte order marks (BOM) are not inserted.

Table 6.3: Supported encodings

local	ansi	oem
wchar_t	utf-8	utf-16
utf-16le	utf-16be	iso-8859-1
iso-8859-2	iso-8859-3	iso-8859-4
iso-8859-5	iso-8859-6	iso-8859-7
iso-8859-8	iso-8859-9	iso-8859-13
iso-8859-15	windows-1250	windows-1251
windows-1252	windows-1253	windows-1254
windows-1255	windows-1256	windows-1257
windows-1258	cp437	cp850
cp866	koi8-r	koi8-u
gb2312	gb18030	big5
shiftjis	euc-jp	euc-kr

6.3.2 *CodecFSM* **type object**

CodecFSM implements a state-based text codec providing conversion from UTF-8 to the specified encoding and vice versa. If a data block ends with an incomplete multibyte sequence, the codec will remember its state and use it later when transcoding the next block. *CodecFSM* objects can be created using the `codec.createcodec()` function.

A simple example:

```
fsm=codec.createcodec("utf-16le")
s=fsm.fromutf8("abc αβγ")
io.write("UTF-16 data: ")
for i=1,#s do
    io.write(string.format("%02X ",s:byte(i)))
end
```

The following data will be printed:

```
UTF-16 data: 61 00 62 00 63 00 20 00 B1 03 B2 03 B3 03
```

codecfsm.close()

Destroys the object.

codecfsm.fromutf8(str)

Converts `str` from UTF-8 to the selected encoding.

Parameters:
    str (*string*): a UTF-8 string to convert

Return value:
    Returns the converted string.

codecfsm.toutf8(str)

Converts `str` from the selected encoding to UTF-8.

Parameters:
    str (*string*): a string to convert

Return value:
    Returns the converted string.

codecfsm.reset([direction])

Resets the codec state.

Parameters:
    direction (*string*, optional): "fromutf8", "toutf8" or "both"; de-
    fault is "both"

6.4 *RegisterMap* **type object**

Unlike most `sdmconsole` GUI objects, *RegisterMap* objects are not created by the `gui` library. Instead, they are produced using the `registermap()` method of the *Channel* type object (Subsection 6.1.5).

`map.pages()`

Return value:

　Returns number of pages in the register map.

`map.addpage([name])`

Adds a new page to the register map.

Parameters:

　name (*string*, optional): page name

Return value:

　Returns the index of the created page (counted from 1).

`map.removepage([index])`

Removes the page with index `index` (counted from 1).

Parameters:

　index (*integer*): index of the page to remove

`map.pagename(index [, name])`

Queries the name of the page `index`, optionally sets it.

Parameters:

　index (*integer*): page index
　name (*string*, optional): a name to set

Return value:

　Returns the previous page name.

`map.rows(index)`

Queries the number of rows for the page `index`.

Parameters:

　index (*integer*): page index

Return value:

Returns the number of rows.

map.insertrow(page, row, data)

Inserts a new row.

Parameters:

page (*integer*): page index
row (*integer*): row index
data (*table*): row data

Remarks:

Row data table contains the following fields:

- For all row types:

  - name (*string*): section, register or FIFO name;

  - type (*string*): "section", "register", "fifo" or "memory".

- For *Register*, *FIFO* and *Memory* type rows:

  - id (*string*): optional identifier (can be absent);

  - addr (*integer*): register address (can be absent);

  - data (*integer* for register, *table* for FIFO or memory): register or FIFO/memory data, can be absent;

  - writeaction (*string*): custom write action (if defined);

  - readaction (*string*): custom read action (if defined);

  - skipgroupwrite (*boolean*): skip this row when writing multiple rows;

  - skipgroupread (*boolean*): skip this row when reading multiple rows.

- Only for *Register* type rows:

  - widget (*string*): widget type for data column ("lineedit", "dropdown", "combobox" or "pushbutton");

  - options (*table*): a table of options for a drop-down list, a combobox or a push button. Table items are themselves tables containing name and value fields.

- Only for *FIFO* and *Memory* type rows:
  - `prewriteaddr` (*integer*): indirect addressing register
  - `prewritedata` (*integer*): indirect addressing data

map`.removerow(page, row)`

Removes an existing row.

Parameters:

    `page` (*integer*): page index
    `row` (*integer*): row index

map`.rowdata(page, row [, data])`

Queries, and optionally sets, data for the specified row.

Parameters:

    `page` (*integer*): page index
    `row` (*integer*): row index
    `data` (*table*, optional): data to set (see the `insertrow()` method description)

map`.currentpage()`

Return value:

    Returns the index of the current page.

map`.currentrow()`

Return value:

    Returns the index of the current (selected) row.

map`.findrow(value, how)`

Searches a row in the register map.

Parameters:

    `value` (*string*): value to search for
    `how` (*string*): type of search: `"name"`, `"id"` or `"addr"`

Return value:

If nothing was found, returns **nil**. If a suitable row was found, returns three values: row data table (see the `insertrow()` method description), page index and row index.

`map.clear()`

Clears the register map.

`map.load(filename)`

Loads register map contents from a file.

Parameters:

filename (*string*): file name

Remarks:

If `filename` is not an absolute path, it will be resolved relative to the current directory or, failing that, the SDM data directory (Section 2.5).

`map.save(filename)`

Saves register map contents to a file.

Parameters:

filename (*string*): file name

6.5   `luadfic` extension module

`luadfic` (Dynamic Function Interface Compiler) is an extension module allowing the user to access functions exported by external shared libraries (`.dll` or `.so`). This feature is sometimes called a *foreign function interface*. `luadfic` is developed as a part of the SDM project, but can be also used with a standalone Lua interpreter or with any program that can use Lua extension modules. Only x86 and x86-64 based machines are currently supported by `luadfic`.

`luadfic` is not loaded automatically; to load it, use the **require**() Lua function.

`luadfic` is a low-level library which by design is able to circumvent most safety mechanisms. Incorrect usage will almost certainly result in program crashes.

The following example imports the `getenv()` function from the Standard C library:

```
dfic=require("luadfic")
getenv=dfic.import(nil, "getenv", "char *@const char *")
path=getenv("PATH")
print(path)
```

The first argument to the `dfic.import()` method is a library name; **nil** means that the symbol will be searched among the libraries already loaded by the current process (this example assumes that the program uses a shared version of the Standard C library).

6.5.1 Function prototypes

The `import()` method has three string parameters: library file name, exported symbol name (on Microsoft Windows, functions can be also imported by a numeric identifier known as *ordinal*) and function prototype. Prototype string can have one of the following formats:

parameters
return_value@parameters
return_value@calling_convention@parameters

parameters is a comma-separated list of parameter types (see below) which can be empty or **void** if a function doesn't have any parameters. *return_value* specifies the return value type (and can be empty or **void** if a function doesn't return a value). *calling_convention* specifies the calling convention; when empty, the default one is used (see Table 6.4).

For functions that have neither parameters nor return value, empty or **nil** prototype string can be supplied.

Types of parameters and return values are specified in the following form:

  [**const**] *fundamental_type* [*]

Conceptually, fundamental types are first-class types that have direct Lua counterparts. Supported fundamental types are listed in Table 6.5. Synonyms are given in parentheses. Lua types for `clock_t` and `time_t` are platform-dependent.

Table 6.4: Supported calling conventions

Name	Description
x86 calling conventions	
cdecl	Default. Arguments are pushed on the stack in reverse order (that is, the first argument has the lowest address), stack cleanup is a responsibility of the caller.
pascal	Arguments are pushed on the stack in direct order, stack cleanup is a responsibility of the callee.
stdcall	Arguments are pushed on the stack in reverse order, stack cleanup is a responsibility of the callee.
fastcall	The first two non-floating point arguments are passed via the ECX and EDX registers, others are pushed on the stack in reverse order, stack cleanup is a responsibility of the callee.
thiscall	The first argument, **this** pointer, is passed via the ECX register, the remaining arguments are pushed on the stack in reverse order, stack cleanup is a responsibility of the callee.
winapi	Equivalent to stdcall.
x86-64 calling conventions	
x64	Calling convention native to the OS. Microsoft x64 ABI is described in *x64 Software Conventions*; Linux uses the ABI defined in the *System V Application Binary Interface AMD64 Architecture Processor Supplement*.
winapi	Equivalent to x64.

Asterisk (*) denotes that a parameter or return value is passed by pointer; the exception is **void** * which is considered a fundamental type by itself (but **void** ** is also a valid type specifier). **const** informs the library that the function is not supposed to alter data referenced by the pointer; it doesn't have meaning for non-pointer types, **void** * or return value types.

6.5.2   Passing data by pointer: implicit buffer allocation

To enable passing data by pointers, luadfic implements an *implicit buffer allocation* mechanism. When invoking a function with pointer parameters (except **void** *), the library will perform the following operations:

1. If the argument is **nil** or a NULL *lightuserdata* (such as dfic.nullptr, see Subsection 6.5.4), pass a NULL pointer.

Table 6.5: Supported fundamental types

DFIC fundamental type	Lua type
`void *`	*lightuserdata*
`bool`	*boolean*
`char` (`unsigned char`, `signed char`)	*string*
`wchar_t`	*integer*
`short int` (`short`, `signed short`, `signed short int`)	*integer*
`unsigned short int` (`unsigned short`)	*integer*
`int` (`signed`, `signed int`)	*integer*
`unsigned int` (`unsigned`)	*integer*
`long int` (`long`, `signed long`, `signed long int`)	*integer*
`unsigned long int` (`unsigned long`)	*integer*
`long long int` (`long long`, `signed long long`, `signed long long int`)	*integer*
`unsigned long long int` (`unsigned long long`)	*integer*
`size_t`	*integer*
`intptr_t`	*integer*
`uintptr_t`	*integer*
`ptrdiff_t`	*integer*
`clock_t`	*integer* or *number*
`time_t`	*integer* or *number*
`int8_t`	*integer*
`uint8_t`	*integer*
`int16_t`	*integer*
`uint16_t`	*integer*
`int32_t`	*integer*
`uint32_t`	*integer*
`int64_t`	*integer*
`uint64_t`	*integer*
`float`	*number*
`double`	*number*

2. Otherwise, allocate a temporary buffer:

   - If the argument has an *integer* or *number* type, allocate a buffer of the specified size and initialize it with zeros.

   - Otherwise:
     - For **char** pointers (also **unsigned char** and **signed char**) the argument must be of *string* type. Allocate a buffer of a size equal to the string size plus one and copy the string to the buffer (appending a terminating null character).
     - For non-**char** pointers the argument must be of *table* type (even if only one value is required). Allocate a buffer of a size equal to the table size and copy the table contents to the buffer.

3. Invoke the function, passing pointers to temporary buffers as arguments.

4. For non-**const**, non-NULL pointers, copy the contents from the buffers and pass them to the user as return values (of *string* type for **char** pointers, of *table* type otherwise). For non-**const** NULL pointers, pass **nil** as a respective return value.

luadfic assumes that **char** pointers reference null-terminated strings. If a fixed buffer size is needed, use explicit buffers (see the next subsection).

Suppose that we want to get a value from the Windows registry. It can be done with the RegGetValueA() function from advapi32.dll which is defined as follows (Windows API macros are decoded in comments):

```
LONG WINAPI RegGetValueA(    /*   long __stdcall    */
    HKEY hkey,               /*   void *            */
    LPCSTR lpSubKey,         /*   const char *      */
    LPCSTR lpValue,          /*   const char *      */
    DWORD dwFlags,           /*   unsigned long     */
    LPDWORD pdwType,         /*   unsigned long *   */
    PVOID pvData,            /*   void *            */
    LPDWORD pcbData          /*   unsigned long *   */
);
```

The "A" suffix in the function name denotes that it uses ordinary narrow-character strings in a locale-dependent "ANSI" encoding (as opposed to UTF-16 encoded wide-character strings). lpSubKey, lpValue, pdwType,

pvData and pcbData parameters are passed by pointers; the last three of
them point to data that are modified by the function.

The arguments that we pass to RegGetValue are:

- 0x80000002 for hKey (HKEY_LOCAL_MACHINE)

- "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion" for
  lpSubKey

- "ProductName" for lpValue

- 2 for dwFlags (restrict value type to null-terminated string)

- **nil** for pdwType (indicate that pdwType is not needed)

- 1024 for pvData (size of the output buffer)

- {1024} for pcbData (initialized with buffer size, contains output
  value size on return)

```
dfic=require("luadfic")

RegGetValue=dfic.import("advapi32.dll", "RegGetValueA",
    "long@winapi@uintptr_t, const char *, const char *, unsigned
    long, unsigned long *, char *, unsigned long *")

r,valtype,value,count=RegGetValue(0x80000002,
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion",
    "ProductName", 2, nil, 1024, {1024})

print("RegGetValue returned "..r)
print("Value is "..value)
print("Value size is "..count[1])
```

Note that the count value is passed and returned as a table, even though
it has only one element.

Pointers returned by external functions are dealt with in a similar way,
with the difference that luadfic doesn't know the size of the buffer ref-
erenced by it. For **char** pointers a null-terminated string is assumed (like
in the example with getenv() in the beginning of the section); otherwise
a single value of the corresponding type will be extracted. If the external
function returns a NULL pointer, luadfic will not try to dereference it,
returning **nil** instead.

6.5.3   Passing data by pointer: explicit buffer allocation

The other method of dealing with pointers involves explicitly allocating buffers on the heap and passing/returning **void** * instead of normal pointers, since **void** * doesn't participate in the implicit buffer allocation described above.

```
dfic=require("luadfic")

RegGetValue=dfic.import("advapi32.dll", "RegGetValueA",
    "long@winapi@uintptr_t, const char *, const char *, unsigned
    long, unsigned long *, void *, unsigned long *")

buf=dfic.buffer(1024)

r,valtype,count=RegGetValue(0x80000002,
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion",
    "ProductName", 2, nil, buf.ptr(), {1024})

print("RegGetValue returned "..r)
print("Value is "..buf.read())
print("Value size is "..count[1])
```

Note that the penultimate parameter type is declared here as **void** * instead of **char** *. For the other pointers implicit buffer allocation is utilized, as in the previous example.

Pointers returned by functions can also be dealt with using explicit buffers. To continue with our getenv() example:

```
dfic=require("luadfic")
getenv=dfic.import(nil, "getenv", "void *@const char *")
ptr=getenv("PATH")
buf=dfic.buffer(ptr,32768)
path=buf.read()
print(path)
```

There are other operations supported by explicit buffers. See Subsection 6.5.6 for details.

Working with structures

Lua 5.3 introduced `string.pack()` and `string.unpack()` functions which can be used together with explicit buffer allocation to work with data structures. In the following example, `string.unpack()` is used to parse the `tm` structure returned by `localtime()`:

```lua
dfic=require("luadfic")

localtime=dfic.import(nil,"localtime","void*@const time_t*")
tmptr=localtime({os.time()})

-- "struct tm" is comprised of 9 integers
buf=dfic.buffer(tmptr,9*dfic.typesize("int"))
tm=table.pack(string.unpack("iiiiiiiii",buf.read(1,#buf)))

print("tm_sec="..tm[1])
print("tm_min="..tm[2])
print("tm_hour="..tm[3])
print("tm_mday="..tm[4])
print("tm_mon="..tm[5])
print("tm_year="..tm[6])
print("tm_wday="..tm[7])
print("tm_yday="..tm[8])
print("tm_isdst="..tm[9])
```

6.5.4 `dfic` global object

`dfic.import(library, symbol [, prototype])`

Creates a wrapper object that can be used to invoke external function exported by `symbol`.

Parameters:

   `library` (*string*): shared library file name (with or without path)

   `symbol` (*string* or *integer*): exported symbol name or ordinal

   `prototype` (*string*, optional): function prototype (syntax is described in Subsection 6.5.1)

Return value:

   Returns a *DficFunction* type object which can be used to invoke the function.

Remarks:

If `library` does not include a path, the dynamic linker will search for the module in standard locations defined by the platform.

If loading the module fails, `luadfic` will try again after appending the standard shared library extension (`"dll"` on Microsoft Windows, `"so"` otherwise) unless `library` already has one. When `library` is either an empty string or **nil**, the requested symbol will be searched among the libraries loaded by the current process.

Import by ordinal is only possible on Microsoft Windows.

On 32-bit Windows, if `symbol` can't be found in the module, `luadfic` will automatically try to apply common C-style name decoration schemes to it, like prepending symbol name with an underscore, @ sign (for the `fastcall` convention) or appending @N suffix (for `fastcall` and `stdcall` conventions).

Note that `luadfic` does not perform C++ name decoration. Functions from C++ libraries can still be imported, but the exact (decorated) symbol name must be supplied.

When `prototype` is omitted, it is assumed that the function has neither parameters nor return value.

dfic.buffer(size)
dfic.buffer(ptr, size)

Creates a *DficBuffer* type object which provides raw memory access. New or existing memory block can be used.

Parameters:

 `size` (*integer*): buffer size
 `ptr` (*lightuserdata*): pointer to the existing buffer

Return value:

 Returns a *DficBuffer* type object.

Remarks:

When invoked with one argument, allocates a memory block on the heap. Allocated memory is zero-initialized. The memory block is owned by the *DficBuffer* object and will be freed when the object is destroyed (either explicitly via the `close()` method or by the Lua garbage collector).

When invoked with two arguments, provides access to an existing buffer pointed by `ptr`. *DficBuffer* object will not try to deallocate this memory.

dfic.info(key)

Provides platform-specific information.

Parameters:

    key (*string*): type on information requested:

        – "os": operating system type ("windows", "unix")

        – "cpu": machine architecture ("x86", "x64")

        – "extension": platform-dependent customary shared library extension ("dll", "so")

Return value:

    Returns a string with the requested information.

dfic.typesize(name)

Parameters:

    name (*string*): name of the parameter or return value type (as in prototype string)

Return value:

    Returns a size of the specified type (in bytes).

dfic.int2ptr(i)

Parameters:

    i (*integer*): integer to convert

Return value:

    Returns a *lightuserdata* value containing pointer represented by i.

dfic.ptr2int(ptr)

Parameters:

    ptr (*lightuserdata*): pointer to convert

Return value:

    Returns an *integer* value representing pointer contained by ptr.

dfic.nullptr

A *lightuserdata* object representing a NULL pointer. Note: this is not a method, but a plain data field. **void** * pointers returned by external func-

tions can be compared with this value. It can be also used as a table element value (table elements can't be **nil** in Lua).

6.5.5   *DficFunction* **type object**

`dficfunction.close()`

Destroys the object. If the library is not used by anything else, it is unloaded from the process address space.

`dficfunction(...)`
`dficfunction.invoke(...)`

Invokes an external function. Parameters are defined by the prototype.

Return value:

Generally returns multiple values. The first is the return value of the external function itself (unless the return value type is empty or **void**). The subsequent return values provide data from buffers pointed by non-**const**, non-**void** pointer arguments.

Remarks:

Note: *DficFunction* type objects override the __call metamethod.

6.5.6   *DficBuffer* **type object**

`dficbuffer.close()`

Destroys the object. If the buffer is owned by the object, it is deallocated.

`dficbuffer.ptr()`

Return value:

Returns a pointer to the underlying memory buffer. The pointer can be used as an argument to the dfic.invoke() method.

`dficbuffer.write(data [, start])`

Writes data to the buffer.

Parameters:

data (*string*): data to write

start (*integer*, optional): first byte to overwrite (counting from 1)

dficbuffer`.read([start, count])`

Reads data from the buffer.

Parameters:

start (*integer*, optional): first byte to read (counting from 1)
count (*integer*, optional): number of bytes to read

Return value:

Returns the read data.

Remarks:

When invoked without parameters, assumes that the buffer holds a null-terminated string and returns that string. If the buffer doesn't contain a null character, returns the whole buffer.

dficbuffer`.resize(newsize)`

Resizes the buffer.

Parameters:

newsize (*integer*): new size

Return value:

Returns a pointer to the resized buffer.

Remarks:

Note: only allocated buffers can be resized.

When the buffer size is increased, the memory block is reallocated and the pointer may change. Data in the buffer are unchanged, any extra space is zero-initialized.

When the buffer size is decreased, the data that fit in the new buffer remain unchanged.

Accessing individual bytes

DficBuffer type objects also override __len, __index and __newindex metamethods, providing convenient access to individual bytes. As in regular Lua arrays, bytes in the buffer are counted from 1. Buffer size can be obtained using Lua length operator (#).

```
dfic=require("luadfic")
buf=dfic.buffer(256)
print(#buf) -- prints "256"
buf.write("Hello, world!")
print(buf[1]) -- prints "H"
buf[5]="X"
buf[6]="\0"
print(buf.read()) -- prints "HellX"
```

6.6 `luaipsockets` extension module

`luaipsockets` library provides Internet Protocol version 4 connectivity based on the most commonly used subset of the Berkeley sockets API. It is developed as a part of the SDM project, but does not depend on other SDM components and can also be used with a standalone Lua interpreter or other programs that can work with Lua extension modules.

`luaipsockets` is not loaded automatically; to load it, use the **require()** Lua function.

A simple example:

```
sockets=require("luaipsockets")

-- Create server
srv=sockets.create("TCP")
srv.setoption("reuseaddr",1)
srv.bind(nil,15555)
srv.listen()

-- Create client
cli=sockets.create("TCP")
cli.connect("127.0.0.1",15555)

-- Server accepts connection
srvconn=srv.accept()

-- Send data from client to server
cli.sendall("Hello from the client!")

-- Client gracefully shuts the connection down
cli.shutdown()
```

```
-- Receive data by the server
r=srvconn.recvall(256)
print("Server received: ["..r.."]")

-- Reply
srvconn.send("Hello from the server!")

-- Server gracefully shuts the connection down
srvconn.shutdown()

-- Receive data by client
r=cli.recvall(256)
print("Client received: ["..r.."]")

-- Close sockets
cli.close()
srvconn.close()
srv.close()
```

6.6.1 **sockets** global object

`sockets.create(protocol)`

Creates a TCP or UDP socket.

Parameters:

   protocol (*string*): protocol ("TCP" or "UDP")

Return value:

   Returns an *IPSocket* type object that can be used to access the socket.

`sockets.gethostbyname(hostname)`

Resolves a host name using the `gethostbyname()` API function. A DNS query will be performed if needed.

Parameters:

   hostname (*string*): a host name to resolve

Return value:

   Returns a string containing the resolved IP address.

sockets.list()

Tries to list the IP addresses of the available network interfaces.

Return value:

   Returns a table of the available network addresses.

6.6.2 *IPSocket* **type object**

***socket*.close()**

Closes the socket and destroys the associated object.

***socket*.bind(address, port)**

Binds the socket to a local address and port.

Parameters:

   address (*string* or **nil**): local IP address
   port (*integer*): local port number

Remarks:

   If address is **nil**, the socket will be bound to all local interfaces (equivalent to the INADDR_ANY constant from the sockets API, or 0.0.0.0). If port is 0, the system will automatically choose an available port.

***socket*.connect(address, port)**

Connects the socket to the remote address and port.

Parameters:

   address (*string*): remote IP address
   port (*integer*): remote port number

Remarks:

   Both TCP and UDP sockets can be connected. For TCP sockets, either connect() or accept() is required to communicate. For UDP sockets connect() is optional, the destination address can be also specified in the send() method. If an UDP socket is connected, it will only receive datagrams from the associated remote.

socket.`listen([backlog])`

Marks the TCP socket as passive, allowing it to accept incoming connections. Returns immediately.

Parameters:

    `backlog` (*integer*, optional): backlog (size of pending connections queue), default value is platform-dependent

socket.`accept()`

Accepts an incoming TCP connection.

Return value:

    Returns an IPSocket type object which can be used to handle the connection.

Remarks:

    This function is blocking. In order to accept incoming connections, `listen()` must be first called.

socket.`send(data [, address, port])`

Sends data through the socket.

Parameters:

    `data` (*string*): data to send
    `address` (*string*, optional): remote IP address
    `port` (*integer*, optional): remote port

Return value:

    Returns the number of bytes sent.

Remarks:

    This function is blocking. For TCP sockets `address` and `port` are ignored, and it is not an error if the number of bytes sent is less than requested.

socket.`sendall(data)`

Sends all the supplied data through the TCP socket.

Parameters:

    `data` (*string*): data to send

Remarks:

This function blocks until all the supplied data are sent, unless an error occurs. Does not return a value. Does not work with UDP sockets.

`socket.recv([n])`

Receives data.

Parameters:

n (*integer*, optional): maximum number of bytes to receive, the default value is large enough to receive an UDP datagram of a maximum size

Return value:

Returns three values:

1. received data (*string*)

2. remote (source) IP address, if available (*string*)

3. remote (source) port, if available (*integer*)

Remarks:

This function is blocking. For TCP sockets, it will return any available data, which can be less than the requested amount. For UDP sockets, the function returns either the full datagram, or n bytes, whatever is less; if n is less than the datagram size, the rest of the datagram is discarded. An empty string as the first return value indicates that the connection was closed.

If the remote address or port are not available (as it is usually the case for TCP), **nil** is returned instead of respective values.

`socket.recvall(n)`

Receives the specified amount of bytes through the TCP socket.

Parameters:

n (*integer*): the number of bytes to receive

Return value:

Returns the received data as a *string*.

Remarks:

This function blocks until the requested number of bytes is read, connection is closed, or an error occurs. Does not work with UDP sockets.

socket.wait([msec [, mode]])

Waits for the socket to become available for input/output operations.

Parameters:

msec (*integer*, optional): number of milliseconds to wait

mode (*string*, optional): type of I/O operation: "r" for reading, "w" for writing, "rw" for both

Return value:

Returns a boolean value: **true** if the socket is available for the requested operation, or **false** otherwise.

Remarks:

This method uses the select() function from the sockets API. If msec is 0, the function does not wait and returns immediately. If msec is -1, it will wait indefinitely until the socket becomes available. Otherwise it will return when either the socket becomes available or the waiting timer elapses. Default value for msec is -1, default value for mode is "r".

socket.shutdown([mode])

Performs a graceful connection shutdown.

Parameters:

mode (*string*, optional): direction of a shutdown: "r" for reading, "w" for writing, "rw" for both, default is "w"

Remarks:

The specified type of operation will no longer be allowed after the shutdown.

Closing the socket can cause loss of queued data that are yet to be sent. In order to perform a graceful TCP connection shutdown, a client should call shutdown("w"), indicating that it does not have any more data to send, and then call the recv() method repeatedly until it returns an empty string.

UDP sockets can also be shut down; however, since the UDP protocol doesn't have a notion of connection, the shutdown will only have a local effect.

socket.setoption(option [, value])

Queries the current socket option value and optionally sets it to the new value.

Parameters:

option (*string*): option name, see the Remarks section below for a list of supported options

value (*integer*, optional): new option value

Return value:

Returns the previous option value.

Remarks:

The following options are supported:

- "broadcast" – allow the socket to send datagrams to a broadcast address. Only supported by UDP sockets.

- "keepalive" – periodically check whether an idle connection is still active by sending keep-alive packets. Only supported by TCP sockets.

- "reuseaddr" – allow the socket to bind to a local address/port combination that has already been bound to.

- "nodelay" – disable Nagle algorithm, that is, send data as soon as possible, even if there is only a small amount of data available. Can lead to inefficient bandwidth usage due to an increased overhead of sending many small packets. Only supported by TCP sockets.

- "sndbuf" – send buffer size, in bytes.

- "rcvbuf" – receive buffer size, in bytes.

For boolean options, a nonzero value means that the option is enabled, otherwise it is disabled.

socket.info()

Queries the local and remote addresses and ports associated with the socket.

Return value:

Returns a table which may contain the following fields:

- localaddr – local IP address

- localport – local port

- remoteaddr – remote IP address

- remoteport – remote port

Some fields can be absent if the information is unavailable.

6.7 `luart` extension module

`luart` library is an extension module providing serial port access capabilities under all operating systems supported by SDM. It does not otherwise depend on SDM and can be used with a standalone Lua interpreter or with any program that can use Lua extension modules.

`luart` is not loaded automatically; to load it, use the **require()** Lua function.

A simple example:

```lua
luart=require("luart")

-- Open port
p=luart.open("COM1")

-- Configure port
p.setbaudrate(115200)
p.setdatabits(8)
p.setstopbits(1)
p.setparity("no")
p.setflowcontrol("no")

-- Write some data
p.write("uname -a\n")

-- Read some data
s1=p.read(8) -- read 8 bytes (blocking)
s2=p.read(8,"part") -- read up to 8 bytes (blocking)
s3=p.read(8,"nb") -- read up to 8 bytes (non-blocking)
s4=p.read("a") -- read all available data (non-blocking)
s5=p.read("l") -- read line (blocking)

-- Close port
p.close()
```

6.7.1 `luart` global object

`luart.open(portname)`

Opens a serial port.

Parameters:

portname (*string*): port name

Return value:

Returns an *Uart* type object which can be used to access the port.

Remarks:

Under Microsoft Windows, `luart` provides multiple ways to reference serial ports:

- Using port names like `COM1`. `luart` will automatically prepend these names with the `"\\.\"` prefix, otherwise ports with numbers larger than `COM9` could not be opened.

- Using the Win32 device namespace: `"\\.\`*name*`"`, like `"\\.\COM10"`. This method can be used for any serial port name.

- Using NT device object names:

  `"\\?\GLOBALROOT\Device\`*devicename*`"`,

  Device object names are not the same as port names; mapping between them is defined in the registry.

Under Linux, `portname` is a device name like `"/dev/ttyS0"` or, for USB serial converters, `"/dev/ttyUSB0"`. In order to open the port, the user must have the necessary permissions, which can require adding the user to the `dialout` or `dialer` group, depending on the distribution.

luart.list()

Tries to list the available serial port names. The results may be not reliable.

Return value:

Returns a table of available serial port names (empty table if no ports were detected).

6.7.2   *Uart* **type object**

***uart*`.close()`**

Closes the port and destroys the object.

***uart*`.setbaudrate([baudrate])`**

Queries the current baud rate and optionally sets it to the new value.

Parameters:

baudrate (*integer,* optional): new baud rate

Return value:

Returns the previous baud rate.

Remarks:

Baud rate support varies depending on hardware, driver and operating system. The following values are fairly typical and likely to work with both Microsoft Windows and most Linux distributions:

110	300	600	1200	2400	4800
9600	19200	38400	57600	115200	

uart.setdatabits([bits])

Queries the current data bits number and optionally sets it to the new value.

Parameters:

bits (*integer,* optional): new number of data bits (5, 6, 7 or 8)

Return value:

Returns the previous data bits number.

uart.setstopbits([bits])

Queries the current stop bits number and optionally sets it to the new value.

Parameters:

bits (*integer,* optional): new number of stop bits (1 or 2)

Return value:

Returns the previous stop bits number.

Remarks:

Note: luart doesn't support Windows-specific value of 1.5 stop bits.

uart.setparity([mode])

Queries the current parity mode and optionally sets it to the new value.

Parameters:

mode (*string,* optional): new parity mode ("no", "even" or "odd")

Return value:

Returns the previous parity mode.

uart`.setflowcontrol([mode])`

Queries the current flow control mode and optionally sets it to the new value.

Parameters:

mode (*string*, optional): new flow control mode (`"no"`, `"hardware"` or `"software"`)

Return value:

Returns the previous flow control mode.

Remarks:

`"hardware"` mode refers to RTS/CTS flow control. `luart` doesn't support DTR/DSR flow control.

uart`.write(data [, mode])`

Writes data to the serial port.

Parameters:

data (*string*): data to write
mode (*string*, optional): write mode (`"all"`, `"part"` or `"nb"`), see the Remarks below

Return value:

Returns the number of bytes written.

Remarks:

In `"all"` mode (the default) the function blocks until either all the provided data have been written, or end-of-file or error condition occurs. In `"part"` mode the function blocks until at least one byte is written. In `"nb"` mode the function doesn't block and returns `0` if no data can be written without blocking.

uart`.read(format [, mode])`

Reads data from the serial port.

Parameters:

format (*integer* or *string*): number of bytes to read (if integer), `"a"` to read all the available data or `"l"` to read until the newline character
mode (*string*, optional): read mode (`"all"`, `"part"` or `"nb"`), see the Remarks below

Return value:

Returns the read data as a string.

Remarks:

When `format` is `"a"`, the operation is always non-blocking. When `format` is `"l"`, the operation is always blocking. `mode` argument is not applicable in either case.

Otherwise, in `"all"` mode (the default) the function blocks until either the requested number of bytes has been read, or end-of-file or error condition occurs. In `"part"` mode the function blocks until at least one byte is read. In `"nb"` mode the function doesn't block and returns an empty string if no data can be read without blocking.

uart`.setdtr(value)`

Sets the DTR (Data Terminal Ready) line status.

Parameters:

`value` (*boolean*): DTR line status

uart`.getdsr()`

Queries the DSR (Data Set Ready) line status.

Return value:

Returns the DSR line status.

uart`.setrts(value)`

Sets the RTS (Request To Send) line status.

Parameters:

`value` (*boolean*): RTS line status

Remarks:

Note: RTS line status can't be set if hardware flow control is active.

uart`.getcts()`

Queries the CTS (Clear To Send) line status.

Return value:

Returns the CTS line status.

6.8 Extending scripting engine

SDM framework uses the standard Lua extension mechanism to load external modules with the **require**() function. Several such modules are shipped with the SDM distribution. Other modules, both off-the-shelf and custom, can be added.

It is important that the extension module is linked against the same Lua shared library as is used by the SDM framework. Under Microsoft Windows, SDM uses the `luaXX.dll` naming convention, where XX is the Lua version number. It is also recommended that the extension module uses the same instance of the Standard C library as SDM and the Lua interpreter, otherwise certain functions, such as passing file handles, will not work.

Under Microsoft Windows, the recommended way of extension deployment is to link the extension module against the Lua library bundled with SDM, preferably with the same toolchain as was used to build SDM itself. Under Linux, better compatibility can be achieved by linking SDM against the Lua library installed from the operating system package repositories (see Subsection 2.4.3).

To create a new Lua module, one must build a shared library which exports a `luaopen_modulename`() function with the following prototype:

```
int luaopen_modulename(lua_State *L);
```

On Microsoft Windows, to ensure that the function is exported, one must either prepend the above declaration with __declspec(dllexport) or use a module definition (DEF) file. For C++ sources, regardless of OS, the above declaration must be also prepended with **extern** "C" linkage specification.

modulename must correspond to the name of the module, where all dots (if any) are replaced with underscores. The function works as any other Lua extension: it should push the table containing library functions on the Lua stack and return 1.

To be loadable from Lua, the module path must match one of the paths defined in the **package.cpath** variable. Default Lua extension directory used by SDM depends on platform and is defined in Table 2.2. If SDM was linked against an external Lua installation (option `OPTION_LUA_SYSTEM`, see Subsection 2.4.3), Lua will also search for loadable modules in system-defined directories.

Note that the Lua library used by SDM is compiled as C (not C++) code. When implementing Lua modules in C++, be aware that Lua error handling functions such as `lua_error()` and `luaL_error()` invoke the `longjmp()`

function from the Standard C library. The C++ standard doesn't require `longjmp()` to call local object destructors, which can lead to resource leaks. The issue can be mitigated by implementing C wrappers for all C++ callbacks, calling `lua_error()` and `luaL_error()` only from these wrappers, and passing error information to them through other means. This approach is used by SDM. In addition to that, make sure that callbacks don't propagate uncaught C++ exceptions by wrapping them in **try**/**catch** blocks; otherwise the exception will bypass the Lua interpreter code, potentially creating resource leaks in the interpreter or leaving it in a broken internal state.

Refer to the Lua documentation for more information about extension modules.

6.8.1 Example

The following example is a trivial Lua extension module that defines one function `hello()` returning the `"Hello, world!"` string:

```
hello.c

#include "lua.h"

#ifdef _WIN32
    #define EXPORT __declspec(dllexport)
#else
    #define EXPORT
#endif

int hello(lua_State *L) {
    lua_pushstring(L,"Hello, world!"); /* push the return value */
    return 1; /* number of return values */
}

EXPORT int luaopen_hello(lua_State *L) {
    lua_newtable(L); /* create and push a new table */
    lua_pushcfunction(L,hello); /* push hello() function */
    lua_setfield(L,-2,"hello"); /* set it as a table element */
    return 1; /* number of return values */
}
```

To build it under Microsoft Windows (with Visual Studio):

```
cl /LD /IINSTALL_PREFIX\include\lua
    INSTALL_PREFIX\lib\lua54.lib hello.c
```

To build it under Microsoft Windows (with MinGW):

```
gcc -shared -IINSTALL_PREFIX\include\lua
    -LINSTALL_PREFIX\lib hello.c -llua54 -o hello.dll
```

To build it under Linux:

```
gcc -shared -fPIC -IINSTALL_PREFIX/include/sdm/lua
    hello.c -o hello.so
```

After being placed to the appropriate directory (see above), the module can be used from Lua as follows:

```
hello=require("hello")
print(hello.hello())
```

Alternatively, when using CMake to build the extension, one can import sdm package using the `find_package` command and link the extension against the `sdm::lua` target. Include directories will be set automatically:

```
cmake_minimum_required(VERSION 3.3.0)
project(hello)
find_package(sdm REQUIRED)
add_library(hello MODULE hello.c)
target_link_libraries(hello sdm::lua)
set_target_properties(hello PROPERTIES PREFIX "")
```

The last line is needed to prevent the `lib` prefix from being added to the module file name.

See also Section 7.7.

Chapter 7

Creating SDM plugins

7.1 Overview

SDM plugin interface provides an abstraction layer allowing the SDM framework to interact with devices without knowing their implementation details. SDM plugins are implemented as shared libraries (on Microsoft Windows – dynamic-link libraries) that can be loaded and unloaded at run time as needed. SDM framework includes an SDK to assist with plugin development.

SDM plugins are not required to provide reentrancy or thread safety guarantees. Multi-threaded client applications must ensure that plugin functions are not accessed simultaneously from multiple threads.

7.2 Application binary interface

SDM API is organized as a set of plain C functions and does not involve passing non-trivial objects such as structures that are likely to depend on the compiler version, build options or the Standard C library implementation. Data types used in the SDM API are consistent with the target platform's data model (see Table 7.1). It is therefore possible to develop plugins in C, C++ or any language that supports creating C-compatible shared libraries and can work with C pointer semantics. SDM framework and plugins can be built with different toolchains and use different runtime libraries.

On x86-based platforms, SDM functions use the *cdecl* calling convention (arguments are passed on the stack in reverse order, stack cleanup is performed by the caller). On other supported architectures, SDM uses the default calling convention defined by the target platform's ABI.

Table 7.1: Data types used in the SDM API (for x86 and x86-64)

Type	Definition	Size x86	Size x86-64	Description
Standard types				
char	—	1	1	
int	—	4	4	
size_t	—	4	8	
pointer	—	4	8	
Additional types defined in `sdmtypes.h`				
sdm_addr_t	uint32_t	4	4	Register address
sdm_reg_t	uint32_t	4	4	Register data
sdm_sample_t	double	8	8	Data stream sample

7.3   Exported functions

All SDM plugins must export the following set of functions:

```
sdmGetPluginProperty
sdmSetPluginProperty
sdmOpenDevice
sdmCloseDevice
sdmGetDeviceProperty
sdmSetDeviceProperty
sdmConnect
sdmDisconnect
sdmGetConnectionStatus
```

Plugins supporting control channels must also export the following set of functions:

```
sdmOpenChannel
sdmCloseChannel
sdmGetChannelProperty
sdmSetChannelProperty
sdmWriteReg
sdmReadReg
sdmWriteFIFO
sdmReadFIFO
```

```
sdmWriteMem
sdmReadMem
```

Plugins supporting data sources must also export the following set of functions:

```
sdmOpenSource
sdmCloseSource
sdmGetSourceProperty
sdmSetSourceProperty
sdmSelectReadStreams
sdmReadStream
sdmReadNextPacket
sdmDiscardPackets
sdmReadStreamErrors
```

Detailed description of SDM API functions is provided in Chapter 9.

Symbol names exported by the plugin module must not be mangled in any way.

7.4 Properties

SDM objects (plugins, devices, channels and sources) support a common property interface. Both property names and values are represented as case-sensitive null-terminated UTF-8 strings. Properties can be writable or read-only. Properties recognized by the SDM framework are listed in Table 7.2. All these properties are optional, although failure to define `Name`, `Devices`, `ConnectionParameters` (if there are any), `Channels` (if channels are supported) or `Sources` (if sources are supported) will result in less intuitive `sdmconsole` user interaction. Plugins are also free to define other properties as needed.

Table 7.2: Properties recognized by SDM

Name	Description
General properties (applicable to all object types)	
Name	Object name
UserScripts	Lists user scripts associated with the object. Used by `sdmconsole`. Format: `"name1,path1,name2,path2..."`

Continuation of the Table 7.2

Name	Description
Plugin properties	
Vendor	Plugin vendor
Devices	Lists names of devices supported by the plugin, ordered by device id
Device properties	
ConnectionParameters	Lists properties that are used as connection parameters by this device
Channels	Lists names of channels supported by the device, ordered by channel id
Sources	Lists names of sources supported by the device, ordered by source id
AutoOpenChannels	If set to open or connect, suggests that all channels should be opened automatically when the device is opened or connected, respectively. This feature is intended for devices with a fixed number of channels.
AutoOpenSources	As above, but for sources
Channel properties	
RegisterMapFile	Path to a file to load register map from. If not an absolute path, it will be resolved relative to the SDM data directory (Section 2.5). Used by sdmconsole.
Source properties	
Streams	Lists names of streams supported by the source, ordered by stream id
ShowStreams	Lists IDs of streams to view by default in sdmconsole
ViewMode	Default representation mode in sdmconsole: bars, plot, bitmap, video or binary. If none is specified, either plot or bars are used, depending on a last user choice

Properties represent a purely software concept. They are not intended to be communicated to the hardware directly; rather, they can affect how the plugin interacts with the hardware.

In addition to ordinary properties, some properties can hold strings that list other strings (e.g. other property names). The string should be in CSV format as defined by RFC 4180[1], where each field represents a list element. Such lists should generally be read-only. There are three predefined lists:

- `*` – lists all properties defined by this object

- `*ro` – lists read-only properties defined by this object

- `*wr` – lists writable properties defined by this object

Any object that defines at least one property must also implement these lists. The predefined lists should not reference themselves.

Properties are accessed using `sdmGet*Property` and `sdmSet*Property` functions.

7.5 Developing SDM plugins in C

SDM plugin should define an `EXPORT_SDM_SYMBOLS` macro and include `sdmapi.h`. The macro must be defined before header inclusion, either through explicit **#define** or via an appropriate compiler option. `sdmapi.h` also includes `sdmtypes.h` which defines types used by SDM (such as `sdm_addr_t`, `sdm_reg_t` and `sdm_sample_t`).

Functions that should be implemented by the plugin are listed in Section 7.3. Their prototypes and semantics are provided in Chapter 9.

SDM SDK includes a simple example, `simpleplugin.c`, which can be used as a reference.

7.6 Developing SDM plugins in C++

Plugins can be developed in C++ as they would in C, with a few caveats:

- exported SDM functions must have an **extern** `"C"` linkage specification (`sdmapi.h` takes care of this);

- exported SDM functions must not propagate uncaught exceptions.

---

[1]`https://www.ietf.org/rfc/rfc4180.txt`

Alternatively, one can use the `pluginprovider` library which is also a part of the SDM SDK. It defines abstract classes that can be used as bases for plugin, device, channel and source classes (Figure 7.1). Their member functions are trivial adapters for SDM API functions and as such are not documented in detail; refer to the Chapter 9. If any of these member functions throws an exception of `std::exception` type, it is caught by the library, the error message is printed to `stdout` and the respective SDM API function returns an error (`SDM_ERROR` or `NULL` as appropriate).

All these classes inherit from `SDMPropertyManager` which implements the common property interface (Section 7.4).

The `pluginprovider` library is not precompiled, but is provided as a set of source files. This makes it possible to build a plugin with any toolchain, not necessarily the same that was used to build the SDM framework itself.

To create a plugin, include `sdmprovider.h` and perform the following steps:

1. Create a plugin class derived from `SDMAbstractPlugin`. Override the `openDevice()` pure virtual function. Define all the necessary properties using `addConstProperty()` for immutable properties and `addProperty()` for editable properties; use `addListItem()` to create lists.

2. Provide a definition for the `instance()` static member function of the `SDMAbstractPlugin` class that returns a pointer to the plugin object instance. It can be done as follows:

```
class TestPlugin : public SDMAbstractPlugin {
// declare TestPlugin members here...
};

SDMAbstractPlugin *SDMAbstractPlugin::instance() {
    static TestPlugin pluginInstance;
    return &pluginInstance;
}
```

In the above example a `TestPlugin` instance is defined as a static local object. This technique guarantees that `instance()` will not return a pointer to an uninitialized object, thus avoiding the situation known as *static initialization order fiasco*.
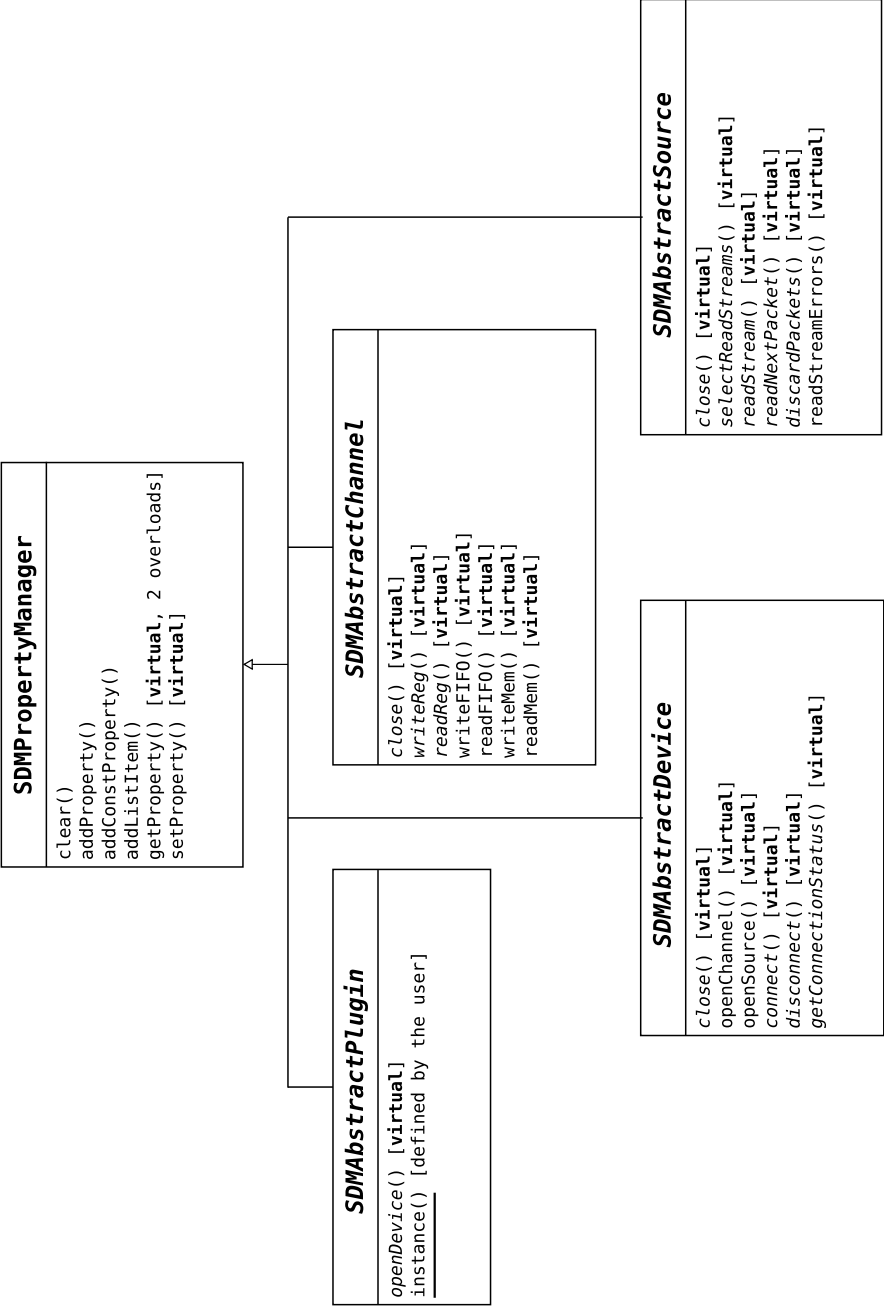
Figure 7.1: `pluginprovider` class diagram

3. Derive device, channel and source classes from the respective abstract classes. Override virtual functions and define the necessary properties as above.

4. Add the `pluginprovider` sources to the build process.

SDM SDK includes an example, `uartdemo`, based on `pluginprovider`. It communicates with an Arduino Uno board over the serial port and provides the means to control pin states and capture data from the integrated ADC.

7.7   Using SDM with CMake package system

CMake implements a package system somewhat akin to *pkg-config*. Libraries using CMake as a build system can facilitate integration into downstream projects by providing a configuration package which can be queried to obtain information about libraries and headers location.

SDM configuration package can be loaded using the `find_package` CMake command. This command makes SDM libraries available as IMPORTED targets with `sdm::` namespace prefix. Libraries exported by SDM are listed in Table 7.3.

Table 7.3: CMake targets exported by SDM

Target	Description
`sdm::api`	SDM API headers
`sdm::pluginprovider`	A C++ library for SDM plugin development
`sdm::lua`	Lua library, defined only when bundled Lua is used
`sdm::ipsockets`	Cross-platform IPv4 sockets library
`sdm::uart`	Cross-platform serial port library

`sdm::api` is an interface library which does not provide object code, but only adds the directory containing SDM API headers to the list of directories searched for include files.

`sdm::ipsockets` and `sdm::uart` are the C++ libraries on which the `luaipsockets` and `luart` bindings are based (see Chapter 6). These libraries are included in the SDK because example plugins depend on them.

For example, `simpleplugin` (see Section 7.5) can be built with the following CMake script:

```
cmake_minimum_required(VERSION 3.3.0)
project(simpleplugin C)
find_package(sdm REQUIRED)
add_library(simpleplugin MODULE simpleplugin.c)
target_link_libraries(simpleplugin sdm::api)
set_target_properties(simpleplugin PROPERTIES PREFIX "")
```

The last line prevents the build system from adding `"lib"` prefix to the output file name when using GCC to ensure consistent plugin naming scheme across all the supported platforms.

For the `find_package` command to work, CMake must know the location of the `sdm-config.cmake` file. There are multiple ways to do it:

- Add the SDM installation prefix to the `CMAKE_PREFIX_PATH` CMake variable, or to the environment variable with the same name.

- Add the SDM installation prefix to the CMake package registry.

- Add the SDM binary directory (Section 2.5) to the `PATH` environment variable.

- Include `sdm-config.cmake` directly using the `include` command instead of `find_package`.

More information is available in the CMake documentation, in particular, the `find_package` command description.

Chapter 8

Creating applications that use SDM plugins

8.1 Overview

In order to use code from the SDM plugin, loading it dynamically is not really necessary – you can just link the plugin code as a static or shared library. If the plugin was created with the `pluginprovider` library, you can even skip the C-style SDM API altogether and access the plugin classes directly.

Still, if you want to load the plugin dynamically, like SDM itself does, SDM sources include reusable code for that purpose. This code is not a part of the SDK. To use it, copy or reference the following libraries to your project and link them to your application:

```
sdk/api
u8e
utils
sdmplug
```

In the application itself, include `sdmplug.h` and use the library classes to access the API functions, as in the following example:

```
main.cpp

#include "sdmplug.h"

#include <iostream>
```

```cpp
int main() {
// Open plugin objects
    SDMPlugin plugin("./testplugin");
    SDMDevice dev(plugin,0);
    SDMChannel ch(dev,0);
    SDMSource src(dev,0);

// Connect the device
    dev.setProperty("Setting1","10.0.0.1");
    dev.setProperty("Setting2","9999");
    dev.connect();

// Write/read a register
    std::cout<<"Writing register..."<<std::endl;
    ch.writeReg(10,0x1234);
    std::cout<<"Reading back... "<<ch.readReg(10)<<std::endl;

// Read some stream data
    std::vector<int> streams {0};
    src.selectReadStreams(streams,0,1);
    sdm_sample_t buf[10000];
    for(int i=0;i<5;i++) {
        auto samples=src.readStream(0,buf,10000);
        std::cout<<"Reading stream 0 data: ";
        std::cout<<samples<<" samples"<<std::endl;
        src.readNextPacket();
    }
}
```

This example application can be built with the following CMake script:

```cmake
cmake_minimum_required(VERSION 3.3.0)
project(sdmtest)
add_subdirectory(u8e)
add_subdirectory(api)
add_subdirectory(utils)
add_subdirectory(sdmplug)
add_executable(sdmtest main.cpp)
target_link_libraries(sdmtest sdmplug)
```

8.2   Notes on the SDMPlug library

SDMPlug class member functions are thin wrappers over the respective SDM API functions and as such are not covered here in detail; refer to Chapter 9.

Plugin paths are assumed to be in UTF-8. If the specified file name does not include an extension, a customary extension for the platform will be appended (`.dll` or `.so`).

Objects owning SDM API handles are reference-counted, so that these handles are not closed until all objects that use them are closed or destroyed. Consider the following example:

```
SDMDevice dev(plugin,0);
SDMChannel ch(dev,0);
dev.close();
ch.writereg(10,0x1234);
ch.close();
```

Here the `dev.close()` statement doesn't result in `sdmCloseDevice()` being called since the device is still used by the `ch` object. Closing the latter, however, causes the device handle to be closed as well.

Chapter 9

SDM API reference

9.1 Plugin functions

```
SDMAPI int SDMCALL sdmGetPluginProperty(const char *name,
    char *buf, size_t n);
```

Copies the requested plugin property value to the buffer pointed by buf of size n bytes.

Parameters:

    name: property name
    buf: buffer to receive property value, can be NULL if n=0
    n: size of the buffer, can be 0

Return value:

    If the function succeeds, it returns 0. If the size of buffer is insufficient, it returns the necessary size (including the terminating null character). If the function fails for some other reason (e.g. no property with such name exists), it returns a negative value.

```
SDMAPI int SDMCALL sdmSetPluginProperty(const char *name,
    const char *value);
```

Sets the plugin property name to value.

Parameters:

    name: property name
    value: property value

117

Return value:

Returns `0` if successful, a non-zero value otherwise.

9.2 Device functions

SDMAPI **void** * SDMCALL sdmOpenDevice(**int** id);

Opens a device object.

Parameters:

id: device id (counting from zero)

Return value:

Returns a device handle if successful, `NULL` otherwise.

Remarks:

It is not specified whether a subsequent call to `sdmOpenDevice()` with the same `id` argument will return a pointer to the same object or a new object. Normally it would be the latter, since it is usually possible to have multiple devices of the same type connected to the same workstation. If `sdmOpenDevice()` returns the same pointer, it must still not destroy the device object until `sdmCloseDevice()` is called the same number of times.

SDMAPI **int** SDMCALL sdmCloseDevice(**void** *h);

Closes the device object.

Parameters:

h: device handle

Return value:

Returns `0` if successful, a non-zero value otherwise.

Remarks:

Note: don't close the device when there are still channels or sources opened on this device.

```
SDMAPI int SDMCALL sdmGetDeviceProperty(void *h, const char
    *name, char *buf, size_t n);
```

Copies the requested device property value to the buffer pointed by `buf` of size `n` bytes.

Parameters:

    `h`: device handle
    `buf`: buffer to receive property value, can be `NULL` if `n=0`
    `n`: size of the buffer, can be `0`

Return value:

    As in `sdmGetPluginProperty`.

```
SDMAPI int SDMCALL sdmSetDeviceProperty(void *h, const char
    *name, const char *value);
```

Sets the device property `name` to `value`.

Parameters:

    `h`: device handle
    `name`: property name
    `value`: property value

Return value:

    Returns `0` if successful, a non-zero value otherwise.

```
SDMAPI int SDMCALL sdmConnect(void *h);
```

Connects to the remote device.

Parameters:

    `h`: device handle

Return value:

    Returns `0` if successful, a non-zero value otherwise.

Remarks:

    If a device requires connection parameters, they are set using the device object's property interface.

SDMAPI **int** SDMCALL sdmDisconnect(**void** *h);

Disconnects from the remote device.

Parameters:

h: device handle

Return value:

Returns 0 if successful, a non-zero value otherwise.

SDMAPI **int** SDMCALL sdmGetConnectionStatus(**void** *h);

Queries the device connection status.

Parameters:

h: device handle

Return value:

Returns a non-zero value for an active connection, zero otherwise.

9.3   Channel functions

SDMAPI **void** * SDMCALL sdmOpenChannel(**void** *hdev, **int** id);

Opens a channel object.

Parameters:

hdev: device handle
id: channel id (counting from zero)

Return value:

Returns a channel handle if successful, NULL otherwise.

Remarks:

It is not specified whether a subsequent call to sdmOpenChannel() with the same id argument will return a pointer to the same object or a new object. If sdmOpenChannel() returns the same pointer, it must still not destroy the object until sdmCloseChannel() is called the same number of times.

```
SDMAPI int SDMCALL sdmCloseChannel(void *h);
```

Closes the channel object.

Parameters:

    h: channel handle

Return value:

    Returns 0 if successful, a non-zero value otherwise.

```
SDMAPI int SDMCALL sdmGetChannelProperty(void *h, const
    char *name, char *buf, size_t n);
```

Copies the requested channel property value to the buffer pointed by buf of size n bytes.

Parameters:

    h: channel handle
    buf: buffer to receive property value, can be NULL if n=0
    n: size of the buffer, can be 0

Return value:

    As in sdmGetPluginProperty.

```
SDMAPI int SDMCALL sdmSetChannelProperty(void *h, const
    char *name, const char *value);
```

Sets the channel property name to value.

Parameters:

    h: channel handle
    name: property name
    value: property value

Return value:

    Returns 0 if successful, a non-zero value otherwise.

```
SDMAPI int SDMCALL sdmWriteReg(void *h, sdm_addr_t addr,
    sdm_reg_t data);
```

Writes to a register. This function is blocking.

Parameters:

    `h`: channel handle

    `addr`: register address

    `data`: data to write

Return value:

    Returns `0` if successful, a non-zero value otherwise.

```
SDMAPI sdm_reg_t SDMCALL sdmReadReg(void *h, sdm_addr_t
    addr, int *status);
```

Reads from a register. This function is blocking.

Parameters:

    `h`: channel handle

    `addr`: register address

    `status`: pointer to a variable to receive operation success status, can be `NULL`

Return value:

    If successful, returns the value read from the register. If `status` is not NULL, sets *`status` to `0` in case of success and to a non-zero value otherwise.

```
SDMAPI int SDMCALL sdmWriteFIFO(void *h, sdm_addr_t addr,
    const sdm_reg_t *data, size_t n, int flags);
```

Writes an array of words to the FIFO.

Parameters:

    `h`: channel handle

    `addr`: register address

    `data`: pointer to an array of words to write

    `n`: number of words to write

    `flags`: zero or any combination of the following:

        – `SDM_FLAG_NONBLOCKING`: perform a non-blocking operation

        – `SDM_FLAG_START`: this operation starts a new packet

Return value:

 If n is 0, this function returns 0.

 In the blocking mode, non-negative value indicates the number of words that has been written, negative value indicates an error. It is not an error when number of words written is less than requested.

 In the non-blocking mode, in addition to the above, SDM_WOULDBLOCK indicates that no data have been written because write operation would be blocking.

Remarks:

 Conceptually, FIFO represents a memory block that is mapped to a single register address. Writing to a FIFO is roughly equivalent to performing a set of ordinary register writes in sequence, but with the support of packets and non-blocking operations.

 In the blocking mode, sdmWriteFIFO() returns when either (1) the requested number of words is written, (2) at least one word is written and no more data can be written at the moment due to lack of free space in FIFO or (3) an error occurs. In the non-blocking mode, in addition, the function returns if no data can be written without blocking.

```
SDMAPI int SDMCALL sdmReadFIFO(void *h, sdm_addr_t addr,
    sdm_reg_t *data, size_t n, int flags);
```

Reads an array of words from the FIFO.

Parameters:

 h: channel handle
 addr: register address
 data: pointer to an array to read to
 n: number of words to read
 flags: zero or any combination of the following:

  – SDM_FLAG_NONBLOCKING: perform a non-blocking operation
  – SDM_FLAG_NEXT: proceed to the next packet

Return value:

 If n is 0, this function returns 0.

 In the blocking mode, returns the number of words that has been read (zero if an end of packet is reached), or a negative value in the case of error. It is not an error when number of words read is less than requested.

In the non-blocking mode, in addition to the above, the function returns `SDM_WOULDBLOCK` when no data have been read because the operation would block.

Remarks:

Conceptually, FIFO represents a memory block that is mapped to a single register address. Reading from a FIFO is roughly equivalent to performing a set of ordinary register reads in sequence, but with the support of packets and non-blocking operations.

If a FIFO supports the notion of packets, `sdmReadFIFO()` will cease reading when all the data from the current packet have been read, regardless of how many words were requested. In order to proceed to the next packet, the function must be called with the `SDM_FLAG_NEXT` flag set. If there are still data from the current packet to be read, they are discarded when `SDM_FLAG_NEXT` flag is supplied. If the read pointer is already at the start of packet, nothing is discarded.

In the blocking mode, `sdmReadFIFO()` will return when either (1) the requested number of words is read, (2) at least one word is read and no more data are currently available for reading, (3) the packet is finished or (4) an error occurs. In the non-blocking mode, in addition, the function returns when no data can be read without blocking.

Note that in the context of FIFO operations, "non-blocking" means that a function will not block due to the FIFO being full or empty. Blocking can still be caused by connection issues. For true non-blocking operations use streams API.

```
SDMAPI int SDMCALL sdmWriteMem(void *h, sdm_addr_t addr,
    const sdm_reg_t *data, size_t n);
```

Writes an array of words to a memory block starting at `addr`.

Parameters:

    `h`: channel handle
    `addr`: register address
    `data`: pointer to an array to write
    `n`: number of words to write

Return value:

    Returns `0` if successful, a non-zero value otherwise.

```
SDMAPI int SDMCALL sdmReadMem(void *h, sdm_addr_t addr,
    sdm_reg_t *data, size_t n);
```

Reads an array of words from a memory block starting at `addr`.

Parameters:

    `h`: channel handle
    `addr`: register address
    `data`: pointer to an array to store read data
    `n`: number of words to read

Return value:

    Returns `0` if successful, a non-zero value otherwise.

9.4 Source functions

```
SDMAPI void * SDMCALL sdmOpenSource(void *hdev, int id);
```

Opens a source object.

Parameters:

    `hdev`: device handle
    `id`: source id (counting from zero)

Return value:

    Returns a source handle if successful, `NULL` otherwise.

Remarks:

    It is not specified whether a subsequent call to `sdmOpenSource()` with the same `id` argument will return a pointer to the same object or a new object. If `sdmOpenSource()` returns the same pointer, it must still not destroy the object until `sdmCloseSource()` is called the same number of times.

```
SDMAPI int SDMCALL sdmCloseSource(void *h);
```

Closes the source object.

Parameters:

    `h`: source handle

Return value:

    Returns `0` if successful, a non-zero value otherwise.

SDMAPI **int** SDMCALL sdmGetSourceProperty(**void** *h, **const char**
  *name, **char** *buf, size_t n);

Copies the requested source property value to the buffer pointed by `buf` of
size `n` bytes.

Parameters:

    `h`: source handle
    `buf`: buffer to receive property value, can be NULL if n=0
    `n`: size of the buffer, can be `0`

Return value:

    As in `sdmGetPluginProperty`.

SDMAPI **int** SDMCALL sdmSetSourceProperty(**void** *h, **const char**
  *name, **const char** *value);

Sets the source property `name` to `value`.

Parameters:

    `h`: source handle
    `name`: property name
    `value`: property value

Return value:

    Returns `0` if successful, a non-zero value otherwise.

SDMAPI **int** SDMCALL sdmSelectReadStreams(**void** *h, **const int**
  *streams, size_t n, size_t packets, **int** df);

Selects data streams for reading.

Parameters:

    `h`: source handle
    `streams`: pointer to an array of stream ids
    `n`: size of the array, can be `0`
    `packets`: suggested minimum number of packets to deliver consecu-
    tively (per stream), `0` for default (see the Remarks below)

df: suggested decimation factor, 1 or greater, see the Remarks below

Return value:

Returns 0 if successful, a non-zero value otherwise.

Remarks:

The streams argument is ignored when n=0 (no streams are selected).

A non-zero packets value is a non-binding request to deliver at least the specified number of consecutive packets after this function is called (for streams that don't support the notion of packets, this argument is interpreted as a number of samples). This information can be used by the plugin, for example, to allocate a buffer of suitable size. The plugin is not required to honor this argument.

A df argument value greater than 1 is a non-binding request to reduce the source output bandwidth by the factor of df. How the decimation is performed, if at all, is up to the implementation: for example, it can deliver only one of each df packets, or deliver all packets, but only one of each df samples in each packet. For high-bandwidth data sources that generate too much data to be processed in real time, it gives the user a choice: either receive decimated data in real time (e.g. for visualisation), or receive complete data for a limited duration (until the buffer is full). This implies some kind of data buffering by the device.

If the stream supports the notion of packets, the next read operation after this function has been called will read from the start of the packet.

Calling this function resets the stream error counter.

```
SDMAPI int SDMCALL sdmReadStream(void *h, int stream,
    sdm_sample_t *data, size_t n, int nb);
```

Reads data from a stream.

Parameters:

h: source handle

stream: stream id

data: pointer to an array to receive data

n: size of the array

nb: a non-zero value indicates a non-blocking operation request

Return value:

In the blocking mode: number of samples read (0 at the end of packet). Negative value indicates an error. It is not an error when number of bytes read is less than requested.

In the non-blocking mode: as above, plus `SDM_WOULDBLOCK` if no data are available for non-blocking reading.

In addition to the above, if `n` is `0`, the functions returns `0`.

Remarks:

In the blocking mode, the function returns when either (1) the requested number of samples has been read, (2) at least one sample has been read and no more data are available for immediate reading, (3) packet ends or (4) an error occurs. In the non-blocking mode, in addition to the above conditions, the function returns when there are no data available for immediate reading.

The requested stream must be selected with `sdmSelectReadStreams()`. The function reads the next data chunk from the stream buffer. To continue reading after the current packet ends, `sdmReadNextPacket()` must be called. If several streams were selected, the respective packets are guaranteed to be delivered synchronously.

The concept of packets is important for stream synchronization if the source has more than one stream. Until `sdmReadNextPacket()` is called, data read from any stream are implied to have been captured at the same time. Packets in different streams can have different sizes. If the stream data aren't inherently packetized, it is still strongly recommended to divide samples into packets for the purpose of synchronization and data visualization.

```
SDMAPI int SDMCALL sdmReadNextPacket(void *h);
```

Proceeds to the next packet. This function affects all streams. Any unread data from the current packet are discarded.

Parameters:

   `h`: source handle

Return value:

   Returns `0` if successful, a non-zero value otherwise.

Remarks:

   If no streams are selected, or the source doesn't support the notion of packets, calling this function has no effect.

SDMAPI **void** SDMCALL sdmDiscardPackets(**void** *h);

Discards all data currently buffered for stream reading. If the source supports the notion of packets, the next read operation will read data from the beginning of the first packet that was added to the buffer after this function was called.

Parameters:

h: source handle

Remarks:

This function resets the stream error counter.

SDMAPI **int** SDMCALL sdmReadStreamErrors(**void** *h);

Gets a number of stream errors for the source.

Parameters:

h: source handle

Return value:

Returns the value of a stream error counter.

Remarks:

A stream error is a condition when consecutive operations read non-consecutive data, possibly because of a packet loss or a receiver buffer overflow.

Appendix A

Command line syntax

A.1 `sdmhost`

```
sdmhost
```

Run `sdmhost` in interactive mode, reading commands from standard input.

```
sdmhost filename [ scriptargs ]
```

Execute a script from the `filename` and exit.

```
sdmhost -i filename [ scriptargs ]
```

Execute a script from the `filename,` then enter interactive mode.

```
sdmhost -h
sdmhost --help
```

Display a short help message.

Script arguments, per Lua custom, are passed in the **arg** global table. The first script argument has index 1. Previous command line arguments have indexes 0 and below.

A.2   `sdmconsole`

```
sdmconsole [ arguments ]
```

Run `sdmconsole`. Optional arguments:

- `--run` *filename* `[` *scriptargs* `]` – execute a script. Must be the last argument (further arguments are passed to the script via the **arg** table, as in `sdmhost`).

- `--batch` *filename* `[` *scriptargs* `]` – execute a script and exit if there were no errors.

Debug options (should not be used normally):

- `--no-redirector` – disable standard output redirection. Lua console will not work properly with this option.

- `--alt-redirector` – under Windows, use alternative standard output redirection method. The alternative method is less intrusive, but also less reliable than the default one. Under Linux this option is ignored.

Appendix B

MHDB 1.2 file format

B.1 Overview

MHDB file format is used by `sdmconsole` to store the captured data streams. It supports multiple channels and high dynamic range samples. MHDB requires a constant packet (line) size and will not be able to store streams with variable packet size. The file consists of three parts:

- Header,

- Metadata,

- Payload.

Metadata block is optional and can be absent.
MHDB uses little-endian format to represent multi-byte values.

B.2 File header

The file header has a fixed size of 32 bytes. Its structure is shown in table B.1.
The header contains the following fields:

- SIGNATURE – `MHDB` signature.

- NLINES – number of lines in the file (per channel).

- NSAMPLES_PER_LINE – number of samples per line.

- CHANNELS – number of channels (1–255).

Table B.1: MHDB header

Offset (bytes)	Bits			
	0–7	8–15	16–23	24–31
0	SIGNATURE			
4	NLINES			
8	NSAMPLES_PER_LINE			
12	CHANNELS	DIGITS	BPS	VERSION
16	META_SIZE		STYPE	*reserved*
20	*reserved*			
24				
28				

- DIGITS – number of significant bits per sample ($\leq 8 \cdot$ BPS for integral sample formats, $= 8 \cdot$ BPS for floating point sample formats).

- BPS – number of bytes per sample, together with STYPE determines the sample format (see the subsection B.3.2 below).

- VERSION – file format version (upper 4 bits – major version number, lower 4 bits – minor version number). The current version (1.2) is represented as 0x12.

- META_SIZE – size of metadata block, in bytes. Must be multiple of 4, can be zero.

- STYPE – sample type, together with BPS determines the sample format (see the subsection B.3.2 below).

Reserved header fields must be filled with zeros. Contents of the metadata block are not specified.

B.3   Payload

The payload consists of lines. Each line is prepended with the 4-byte line header. Lines that were received simultaneously from multiple channels are written sequentially in the ascending order of channel numbers. The total line size in bytes (including header):

$$\text{NSAMPLES\_PER\_LINE} \cdot \text{BPS} + 4$$

B.3.1 Line header

Table B.2: MHDB line header format

Offset (bytes)	Bits			
	0–7	8–15	16–23	24–31
0	SEQ			CH

Line header format is shown in table B.2. Line header contains the following fields:

- SEQ – line number, counting from 0. Lines that are received simultaneously (from different channels) have the same number.

- CH – channel number, counting from 0.

B.3.2 Sample format

The line header is followed by data samples. Number of samples is determined by the NSAMPLES_PER_LINE field of the file header. The sample format is determined by BPS and STYPE fields. Supported formats are listed in table B.3.

Table B.3: Supported MHDB sample formats

BPS	STYPE	Format
1	0	8-bit unsigned integer
1	1	8-bit signed integer
2	0	16-bit unsigned integer
2	1	16-bit signed integer
4	0	32-bit unsigned integer
4	1	32-bit signed integer
4	2	32-bit floating point (single precision)
8	0	64-bit unsigned integer
8	1	64-bit signed integer
8	2	64-bit floating point (double precision)