

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

H.766

(08/2018)

SERIES H: AUDIOVISUAL AND MULTIMEDIA SYSTEMS

IPTV multimedia services and applications for IPTV –
IPTV multimedia application frameworks

Lua for IPTV services

Recommendation ITU-T H.766

ITU-T H-SERIES RECOMMENDATIONS
AUDIOVISUAL AND MULTIMEDIA SYSTEMS

CHARACTERISTICS OF VISUAL TELEPHONE SYSTEMS	H.100–H.199
INFRASTRUCTURE OF AUDIOVISUAL SERVICES	
General	H.200–H.219
Transmission multiplexing and synchronization	H.220–H.229
Systems aspects	H.230–H.239
Communication procedures	H.240–H.259
Coding of moving video	H.260–H.279
Related systems aspects	H.280–H.299
Systems and terminal equipment for audiovisual services	H.300–H.349
Directory services architecture for audiovisual and multimedia services	H.350–H.359
Quality of service architecture for audiovisual and multimedia services	H.360–H.369
Telepresence, immersive environments, virtual and extended reality	H.420–H.439
Supplementary services for multimedia	H.450–H.499
MOBILITY AND COLLABORATION PROCEDURES	
Overview of Mobility and Collaboration, definitions, protocols and procedures	H.500–H.509
Mobility for H-Series multimedia systems and services	H.510–H.519
Mobile multimedia collaboration applications and services	H.520–H.529
Security for mobile multimedia systems and services	H.530–H.539
Security for mobile multimedia collaboration applications and services	H.540–H.549
VEHICULAR GATEWAYS AND INTELLIGENT TRANSPORTATION SYSTEMS (ITS)	
Architecture for vehicular gateways	H.550–H.559
Vehicular gateway interfaces	H.560–H.569
BROADBAND, TRIPLE-PLAY AND ADVANCED MULTIMEDIA SERVICES	
Broadband multimedia services over VDSL	H.610–H.619
Advanced multimedia services and applications	H.620–H.629
Ubiquitous sensor network applications and Internet of Things	H.640–H.649
IPTV MULTIMEDIA SERVICES AND APPLICATIONS FOR IPTV	
General aspects	H.700–H.719
IPTV terminal devices	H.720–H.729
IPTV middleware	H.730–H.739
IPTV application event handling	H.740–H.749
IPTV metadata	H.750–H.759
IPTV multimedia application frameworks	H.760–H.769
IPTV service discovery up to consumption	H.770–H.779
Digital Signage	H.780–H.789
E-HEALTH MULTIMEDIA SYSTEMS, SERVICES AND APPLICATIONS	
Personal health systems	H.810–H.819
Interoperability compliance testing of personal health systems (HRN, PAN, LAN, TAN and WAN)	H.820–H.859
Multimedia e-health data exchange services	H.860–H.869
Safe listening	H.870–H.879

For further details, please refer to the list of ITU-T Recommendations.

Recommendation ITU-T H.766

Lua for IPTV services

Summary

Recommendation ITU-T H.766 specifies Lua as one of the standard multimedia application frameworks for Internet protocol television (IPTV) services. Lua is an extension programming language designed to support general procedural programming with data description facilities. Lua is the scripting language for the Ginga-nested context language (Ginga-NCL) in Recommendation ITU-T H.761. The Lua engine is distributed as free software under the MIT licence.

Lua for IPTV services is structured in two sets of application programming interfaces (APIs). The Lua IPTV core API is a basic, mandatory API that conforms with Ginga-NCL in Recommendation ITU-T H.761. The Lua IPTV extended API includes enhanced functionality that a Lua implementation is recommended to support. Recommendation ITU-T H.766 specifies the Lua IPTV core API and the Lua IPTV extended API. Recommendation ITU-T H.766 includes an annex with a reference manual for Lua 5.1 as well as an appendix with sample Lua code that implements a simple health application.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T H.766	2018-08-29	16	11.1002/1000/13671

* To access the Recommendation, type the URL <http://handle.itu.int/> in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2019

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Table of Contents

		Page
1	Scope	1
2	References.....	1
3	Definitions	1
	3.1 Terms defined elsewhere	1
	3.2 Terms defined in this Recommendation	1
4	Abbreviations and acronyms	2
5	The Lua scripting language.....	2
6	Lua restricted functionalities	3
	6.1 Functions removed from Lua library	3
7	Execution model.....	3
8	Lua IPTV Core API.....	3
	8.1 The canvas module	3
	8.2 The event module	4
	8.3 The persistent module	4
	8.4 The security module	4
	8.5 Lua event classes	4
	8.5.1 The key class	4
	8.5.2 The pointer class	4
	8.5.3 The udp class	4
	8.5.4 The tcp class	5
	8.5.5 The http class	5
	8.5.6 The metadata class	5
	8.5.7 The user class	5
9	Lua IPTV Extended API.....	5
	9.1 The pvr module.....	6
	9.1.1 Constructors.....	6
	9.1.2 Attributes	6
	9.1.3 Primitives	6
	9.2 The widgets module.....	7
	9.2.1 Object widget.....	7
	9.2.2 Object textinput	9
	9.2.3 Object cursor	10
	9.3 Lua event classes	12
	9.3.1 Extended tcp class:.....	12

	Page
9.3.2 Extended udp class:	13
9.3.3 The rtp class.....	14
9.3.4 The sms class	14
9.3.5 The sds class	15
9.3.6 The smartcard class.....	19
9.3.7 The pvr class.....	20
9.3.8 The widget class	20
Annex A – Lua Specification.....	22
A.1 The language	22
A.1.1 Used Notation	22
A.1.2 Lexical conventions	22
A.1.3 Values and types	23
A.1.4 Variables	24
A.1.5 Statements	25
A.1.6 Expressions.....	28
A.1.7 Visibility rules	34
A.1.8 Error handling.....	34
A.1.9 Metatables	34
A.1.10 Environments.....	39
A.1.11 Garbage collection	39
A.1.12 Coroutines	40
A.2 The application program interface.....	41
A.2.1 Basic Concepts	41
A.2.2 The stack	41
A.2.3 Stack size.....	42
A.2.4 Pseudo-indices	42
A.2.5 C Closures	42
A.2.6 Registry	42
A.2.7 Error handling in C	43
A.2.8 Functions and types	43
A.2.9 The debug interface	58
A.3 The auxiliary library	61
A.3.1 Basic Concepts	61
A.3.2 Functions and types	61
A.4 Standard libraries.....	68
A.4.1 Overview	68
A.4.2 Basic functions	68
A.4.3 Coroutine manipulation.....	73

	Page
A.4.4 Modules.....	74
A.4.5 String manipulation.....	76
A.4.6 Table manipulation	80
A.4.7 Mathematical functions.....	81
A.4.8 Input and output facilities.....	81
A.4.9 Operating system facilities	85
A.4.10 The debug library.....	86
A.5 Lua Stand-alone.....	88
A.6 Incompatibilities with the version 5.0.....	89
A.6.1 Changes in the language.....	90
A.6.2 Changes in the libraries.....	90
A.6.3 Changes in the API	90
A.7 The complete syntax of Lua	90
Appendix I – Lua Scripting examples.....	92
I.1 Example using canvas and event modules	92
Bibliography	95

Recommendation ITU-T H.766

Lua for IPTV services

1 Scope

This Recommendation specifies the Lua scripting language to provide interoperability and harmonization among Internet protocol television (IPTV) multimedia application frameworks. To provide global standard IPTV services, it is foreseeable that a combination of different standard multimedia application frameworks will be used. Therefore, this Recommendation specifies the Lua language as one of those standard multimedia application frameworks to provide interoperable use of IPTV services.

Lua for IPTV services is structured in two application programming interfaces (APIs). The Lua IPTV core API is a basic, mandatory API that conforms to the Gingga-nested context language (Ginga-NCL) [ITU-T H.761], which is a presentation engine that integrates NCL and Lua players into a declarative environment. The Lua IPTV extended API includes enhanced functionality that a Lua implementation is recommended to support.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this document. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this document are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

- | | |
|---------------|--|
| [ITU-T H.761] | Recommendation ITU-T H.761 (2014), <i>Nested context language (NCL) and Gingga-NCL</i> . |
| [ITU-T H.770] | Recommendation ITU-T H.770 (2015), <i>Mechanisms for service discovery and selection for IPTV services</i> . |

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

3.1.1 application programming interface (API) [b-ITU-T J.200]: Software libraries that provide uniform access to system services.

3.1.2 scripting language [ITU-T H.761]: The language used to describe an imperative object content that is embedded in another host language. For example, Lua is a scripting language for NCL documents as ECMAScript is a scripting language for HTML documents.

3.2 Terms defined in this Recommendation

This Recommendation defines the following terms:

3.2.1 Lua event class: A group of events that can be generated or handled by Lua scripts. Each group is usually related to a specific functionality of an Internet protocol television (IPTV) terminal device.

3.2.2 Lua Internet protocol television core application programming interface: The set of Lua functions and event classes that a Lua implementation is required to support for basic IPTV functionality.

3.2.3 Lua Internet protocol television extended application programming interface: The set of Lua functions and event classes that a Lua implementation is recommended to support for extended IPTV functionality.

3.2.4 Lua module: A group of functions that can be used by Lua scripts. Each group is usually related to a specific functionality of an IPTV terminal device.

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
BNF	Backus-Naur Form
HTTP	Hypertext Transfer Protocol
IPTV	Internet Protocol Television
NCL	Nested Context Language
PVR	Personal Video Recorder
RTP	Real-time Transport Protocol
SDS	Service Discovery and Selection
SMS	Short Message Service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

5 The Lua scripting language

Lua is an extension programming language designed to support general procedural programming with data description facilities. It also offers good support for object-oriented programming, functional programming and data-driven programming. Lua is intended to be used as a lightweight scripting language for any program that needs one. Lua is implemented as a library, written in clean C (i.e., in the common subset of ANSI C and C++).

Being an extension language, Lua has no notion of a "main" program: it only works embedded in a host client, called the embedding program or simply the host. This host program may invoke functions to execute a piece of Lua code, may write and read Lua variables, and may register C functions to be called by Lua code. Through the use of C functions, Lua may be augmented to cope with a wide range of different domains, thus creating customized programming languages sharing a syntactical framework. The Lua distribution includes a sample host program called lua, which uses the Lua library to offer a complete, stand-alone Lua interpreter.

Annex A presents the Lua specification as a general-purpose language. However, any conformant implementation of Lua for IPTV services shall follow the restrictions specified in clause 6 and shall provide the Lua IPTV core API described in clause 8.

The Lua IPTV extended API described in clause 9, in its turn, is optional, but shall be followed if any of its functionalities is to be implemented.

6 Lua restricted functionalities

6.1 Functions removed from Lua library

The following functions are platform dependent and shall be removed from Lua libraries (see clause A.4) in conformant implementations:

- 1) from module *package*: *loadlib*;
- 2) from module *os*: *clock*, *execute*, *exit*, *getenv*, *remove*, *rename*, *tmpname* and *setlocale*;
- 3) from module *debug*: all functions.

7 Execution model

Lua, as an extension programming language, is intended to be used as a powerful, lightweight scripting language for any program that needs one. Lua scripts are commonly used as a piece of code embedded into another program, written in another language. Therefore, Lua scripts may have different execution models, depending on the host program interpreter. Refer to the specification of the host program interpreter (e.g. Ginga-NCL [ITU-T H.761]) for information about the Lua script lifecycle and available controlling interfaces.

8 Lua IPTV core API

The Lua IPTV core API includes all basic functionality a conformant implementation shall provide. Besides the Lua standard library (restricted as described in clause 6) the following Lua modules shall be present and automatically loaded in a conformant implementation:

- 1) *canvas*: offers an API to draw basic graphical components and manipulate images;
- 2) *event*: allows Lua applications to communicate with the middleware through events;
- 3) *persistent*: exports a table with persistent variables that may be manipulated only by imperative objects.
- 4) *security module*: responsible for offering functionalities related to data security.

The following event classes shall be supported in a conformant implementation:

- 1) *key class*: allows for handling of events generated by input devices;
- 2) *pointer class*: allows for handling of events generated by pointer devices, e.g., a mouse;
- 3) *tcp, udp and http classes*: allows for event-driven communication using the transmission control protocol (TCP), user datagram protocol (UDP) and hypertext transfer protocol (HTTP), respectively;
- 4) *metadata class*: allows for event-driven access to metadata information;
- 5) *user class*: allows applications to extend their functionality by creating their own events.

8.1 The canvas module

A canvas offers a graphical API to be used in a Lua application. Using this API, it is possible to draw lines, rectangles, text, images, etc.

A canvas keeps in its state a set of attributes under which the drawing primitives operate. For instance, if its colour attribute is blue, a call to `canvas:drawLine()` will draw a blue line on the canvas.

A global variable called *canvas* is available after the initialization of a Lua application, representing a graphic region with size and coordinates equal to the display information given to the Lua interpreter.

The canvas module is mandatory for the Lua IPTV core API and must be in conformance with the canvas module as specified in clause 10.3.1 of [ITU-T H.761].

8.2 The event module

This module offers an API for event handling. Using the API, a Lua application may communicate with many middleware components asynchronously.

An application may also use this mechanism internally, using the "user" event class.

The typical use of Lua application is to handle events: Middleware-generated events or events coming from user interactions (e.g., through the remote control).

During its initiation, before becoming event oriented, a Lua script must register an event handler function. After the initialization, any action performed by the script will be in response to an event notified to the application, i.e., to the event handler function.

The event module is mandatory for the Lua IPTV core API and must be in conformance with the event module as specified in clause 10.3.2 of [ITU-T H.761].

8.3 The persistent module

Lua applications may save data in a restricted middleware area and recover it between executions. The Lua player shall allow an application to persist a value to be used by itself or by another imperative object. In order to do that it defines a reserved area, inaccessible to non-imperative objects. This area is split into the groups "service", "channel" and "shared". There are no predefined or reserved variables in these groups, and imperative objects are allowed to change variable values directly. Other imperative languages should offer an API to access this same area.

In this module, Lua offers an API to export the *persistent* table with the variables defined in the reserved area.

The persistent module is mandatory in the Lua IPTV core API and must be in conformance with the persistent module as specified in clause 10.3.4 of [ITU-T H.761].

8.4 The security module

The objects that are part of the security module are responsible for offering functionalities related to data security, e.g., digital signature generation and verification, message digest generation, as well as mechanisms for data encryption.

The security module is mandatory in the Lua IPTV core API and must be in conformance with the security module as specified in clause 11.2 of [ITU-T H.761].

8.5 Lua event classes

8.5.1 The key class

The key class allows for handling of events generated by input devices.

The key class is an event class mandatory in the Lua IPTV core API and must be in conformance with the key class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.2 The pointer class

The pointer class allows for handling of events generated by pointer devices, such as mouse.

The pointer class is an event class mandatory in the Lua IPTV core API and must be in conformance with the pointer class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.3 The udp class

The udp class allows event-driven communication using the UDP.

The udp class is an event class mandatory in the Lua IPTV core API and must be in conformance with the udp class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.4 The tcp class

The tcp class allows event-driven communication using the TCP.

The tcp class is an event class mandatory in the Lua IPTV core API and must be in conformance with the tcp class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.5 The http class

The http class allows event-driven communication using the HTTP.

The http class is an event class mandatory in the Lua IPTV core API and must be in conformance with the http class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.6 The metadata class

The metadata class allows event-driven access to metadata information.

The metadata class is an event class mandatory in the Lua IPTV core API and must be in conformance with the metadata class as specified in clause 10.3.2.2 of [ITU-T H.761].

8.5.7 The user class

The user class allows applications to extend their functionality by creating their own events.

The user class is an event class mandatory in the Lua IPTV core API and must be in conformance with the user class as specified in clause 10.3.2.2 of [ITU-T H.761].

9 Lua IPTV extended API

The Lua IPTV extended API includes enhanced functionality a conformant implementation may provide. If implemented, the functionality described in this clause must be in conformance with the following API specification.

The Lua IPTV extended API includes the following Lua modules:

- 1) pvr module: offers an API to control the personal video recorder (PVR) functionality;
- 2) widgets module: offers an API to draw graphical widgets like buttons and input text boxes.

The following event classes shall be supported in a conformant implementation:

- 1) extended tcp class: allows for extended, event-driven communication using the TCP;
- 2) extended udp class: allows for extended, event-driven communication using UDP protocol;
- 3) rtp class: allows for event-driven communication using the real-time transport protocol (RTP);
- 4) sms class: allows for event-driven communication using the short message service (SMS) protocol;
- 5) sds class: allows applications to receive events from the service discovery and selection (SDS) [ITU-T H.770] functionality;
- 6) smartcard class: allows for handling of events generated by smartcard devices;
- 7) pvr class: allows for handling of events generated by PVR operations;
- 8) widget class: allows for handling of events generated by graphical widgets.

Lua modules and event classes mentioned in the foregoing are detailed in the following clauses. The definition of each function uses the following naming convention:

```
funcname (parnameI: partypeI [; optnameI: opttypeI]) -> retname: rettype
```

9.1 The pvr module

The pvr module offers support for PVR control for IPTV systems.

9.1.1 Constructors

pvr:new (filename: string) -> pvr: object

Arguments

Filename: Filename path (audio/video stream)

Return values

pvr New pvr object

Description

Returns a new pvr object, which is used to control the PVR functionality. If an error occurs, the pvr object is returned with the attribute attrError different from 0 and all attempts to call methods of this object will return the error PVR_ERROR_CREATE.

9.1.2 Attributes

pvr:attrError() -> errorCode: number

Arguments

none

Return values

errorCode – The error code. The possible values are:

- 1 - PVR_ERROR_CREATE
- 2 - PVR_ERROR_DEVICE_NOT_PRESENT
- 3 - PVR_ERROR_PERMISSION_DENIED
- 4 - PVR_ERROR_NOT_ENOUGH_SPACE
- 5 - PRV_ERROR_UNKOWN_SERVICEID
- 6 - PVR_ERROR_INVALID_DATE_TIME
- 7 - PVR_ERROR_INVALID_OPERATION_ID
- 8 - PVR_ERROR_CANCELLING_OPERATION
- 10 - PVR_ERROR_UNKOWN

Description

Returns an error code representing an error in the last PVR operation

9.1.3 Primitives

pvr:schedule ([datetime:table]; [deviation], duration, serviceId: number) -> operationId: number

Arguments

datetime: the recording start time, as an absolute date/time. It is a table value with the following fields: year, month (1-12), day (1-31), hour (0-23), min (0-59), sec (0-59), and isdst (a daylight saving flag, boolean). If omitted, it means immediately.

deviation: number of seconds to anticipate (negative number) or defer (positive number) the recording start time. If omitted or equals to 0 (zero), it means no time deviation.

duration: duration of recording operation in seconds

serviceId: service Id that shall be recorded

<i>Return values</i> operationId representing the pvr record. It may also indicate an error if the operation failed. The possible errors are: <pre> PVR_ERROR_INVALID_DATE_TIME PVR_ERROR_UNKNOWN_SERVICEID PVR_ERROR_PERMISSION_DENIED PVR_ERROR_UNKOWN </pre> <i>Description</i> Schedules a PVR recording operation. If datetime and deviation are omitted, then recording starts immediately. This can be also used for time shifting operation where the playback media becomes the recorded file instead of the received transport stream.
pvr:getList() -> pvrList: table <i>Return values</i> Table containing all active pvr operations <pre> pvrList = { { pvrOperationId = <number>, startDateTime = <table>, duration = <number>, serviceId = <number>, }, ... } </pre> <i>Description</i> Lists all scheduled PVR operations. startDateTime represents the effective recording start time, considered a specified deviation (see pvr:schedule()).
pvr:cancel (pvrOperationId: number) -> errorCode: number <i>Arguments</i> pvrOperationId – id of the pvr operation to be cancelled <i>Return values</i> errorCode representing an error in the cancel operation. The possible values are: <pre> PVR_ERROR_INVALID_OPERATION_ID PVR_ERROR_CANCELLING_OPERATION PVR_ERROR_PERMISSION_DENIED PVR_ERROR_UNKNOWN </pre> <i>Description</i> Cancel a scheduled PVR operation

9.2 The widgets module

The widgets module offers drawing primitives to facilitate the design of richer graphical components, such as buttons and input text boxes, which are not supported by the Lua canvas module. This module also offers cursor handling support and mechanisms for creating and scrolling a virtual canvas, which is a canvas with a larger area than that defined for a Lua application canvas.

9.2.1 Object widget

The widget object is responsible for filling an optional given canvas with a graphic element (widget) and also for posting input events to the registered handlers.

9.2.1.1 Constructors

widget:new (canvas: object)-> widget: object

Arguments:

canvas – a canvas object to be associated with this widget object. This canvas can have an area larger than that defined for its Lua application, therefore it is possible to scroll over it.

Return values:

widget – a new widget object

Description

Returns a new widget object.

9.2.1.2 Attributes

widget:attrCanvas(canvas: object)

Arguments:

canvas – Sets the canvas object used to display the graphic element of this widget.

Description

Sets the canvas object related to this widget.

widget:attrCanvas()->canvas: object

Return Values:

canvas – a canvas object that will be related to this widget.

Description

Retrieves the associated canvas object of this widget.

9.2.1.3 Miscellaneous

widget:scroll(x,y: number)

Arguments:

x,y – number values used for scrolling the associated canvas.

Description

Scrolls the widget canvas to a given position (identified by two integer values, x and y) by composing it along with the application canvas.

widget:addTooltip(x,y,w,h: number, msg: string)-> tooltipID: number

Arguments:

x,y,w,h – number values defining the area of activation for this tooltip.

msg – a string containing the tooltip message.

Return Values:

tooltipID – A unique number value for this tooltip identification

<i>Description</i> Adds a tooltip to the widget, to be displayed when the cursor stops over the specified area.
widget:removeTooltip(tooltipID: number) <i>Arguments:</i> tooltipID – a unique number value identifier for a tooltip, generated by addTooltip function. <i>Description</i> Remove the tooltip identified by the value of tooltipID.

9.2.2 Object textinput

The textinput object is responsible for offering a mechanism for capturing text from the user input, so that it can be eventually dispatched to a widget object. The textinput object extends the event module input handling by adding support for modifiers keys (e.g. CTRL+C, SHIFT+A).

9.2.2.1 Constructors

textinput:new ()-> textinput: object <i>Return values:</i> textinput – a new textinput object <i>Description</i> Returns a new textinput object, which is a graphical component that allows users to input text. This object will dispatch the events generated by user input interaction to the registered handlers.
textinput:new (canvas: object)-> textinput: object <i>Arguments:</i> canvas – the canvas where the widget graphical component will be rendered <i>Return values:</i> textinput – a new textinput object <i>Description</i> Returns a new textinput object that allows a Lua application to handle text input. This object will dispatch the events generated by user input interaction to registered handlers. The canvas object parameter is used to display the captured text. The textinput will be formatted according to the attributes of its canvas, e.g., font (attrFont) or color (attrColor).

9.2.2.2 Attributes

textinput:attrCanvas(canvas: object) <i>Arguments:</i> canvas – Sets the canvas object to be associated with this textinput, so that it is able to display the entered text on this canvas. <i>Description</i> Sets the canvas object to be associated with this textinput, in order to display the entered text on the canvas.

textinput:attrCanvas()->canvas: object <i>Return Values:</i> canvas – a canvas object to be associated with this textinput. <i>Description</i> Retrieves the associated canvas object of this textinput.
textinput:attrText(text: string) <i>Arguments:</i> text – a string containing the text to fill the textinput. <i>Description</i> Sets a text that fills the textinput space, which can later be changed by the user.
textinput:attrText()->text: string <i>Return Values:</i> text – a string containing the textinput current text. <i>Description</i> Returns the current text filled in this object.
textinput:attrEnable(enable: boolean) <i>Arguments:</i> enable – a boolean value to define whether this widget is active. <i>Description</i> Sets the status of this widget to active (able to be filled by the user) or inactive (blocked for user input).
textinput:attrEnable()->enable: Boolean <i>Return Values:</i> enable – a boolean with the current activation status of this widget. <i>Description</i> Returns the current activation status of this widget.

9.2.3 Object cursor

A cursor object is a pointer cursor (associated with an image that is used as its graphical representation), which is exhibited over a canvas.

When the supporting platform is provided with a pointing device, one instance of this object is automatically created for a Lua application. Lua applications can independently customize this object (e.g., change the cursor image) while inside its exhibiting canvas.

9.2.3.1 Attributes

cursor:attrImage (path: string) <i>Arguments:</i> path – a string containing the image source path. <i>Description</i> Sets the source URL of a cursor image to a given string value, replacing the cursor image.
cursor:attrImage()->path: string <i>Return Values:</i> path – the source path string for the image being used as a pointer cursor. <i>Description</i> Returns the current source path for the cursor image.
cursor:attrPosition(x,y: number) <i>Arguments:</i> x,y – two numbers to define the new positioning of a cursor in the canvas. <i>Description</i> Sets the position of the cursor to a given coordinate (expressed by the x and y numbers) in the canvas. A <code>WIDGET_CURSOROVER</code> event is launched after cursor positioning.
cursor:attrPosition()->x,y: number <i>Return Values:</i> x,y – two numbers containing the current coordinates of a cursor in the canvas. <i>Description</i> Returns the current coordinates of the cursor in the canvas.
cursor:attrEnable(enable: boolean) <i>Arguments:</i> enable – a boolean defining whether the cursor is active. <i>Description</i> Sets the status of the cursor to active or inactive. An active cursor is rendered on the screen according to its attributes and allows for user. An inactive cursor is hidden.
cursor:attrEnable()->enable: Boolean <i>Return Values:</i> enable – a boolean with the current status of the cursor. <i>Description</i> Returns the current status of the cursor (active or inactive)

9.2.3.2 Miscellaneous

cursor:click(button: number)

Arguments:

button – a number value identifying the button to be clicked. See the possible values in clause 9.3.8.

Description

Performs a single click on the current position the cursor is over and generates two events of the class 'widget', type cursor, subtypes 'cursor_press' and 'cursor_release'. See clause 9.3.8.

cursor:press(button: number)

Arguments:

button – a number value identifying the button to be pressed. See the possible values in clause 9.3.8.

Description

Presses the specified button on the current position the cursor is over and generates an event of the class 'widget', type cursor, subtype 'cursor_press'. See clause 9.3.8.

cursor:release(button: number)

Arguments:

button – a number value identifying the button to be released. See the possible values in clause 9.3.8.

Description

Releases the specified button and generates an event of the class 'widget', type cursor, subtype 'cursor_release'. See clause 9.3.8.

9.3 Lua event classes

9.3.1 Extended tcp class:

The Lua extended tcp class allows for event-driven communication using the TCP

In order to send or receive TCP data in client-mode, a connection shall be firstly established through posting an event in the form:

```
evt = { class='tcp', type='connect', host=string, port=number,  
        [interface_addr=string], [local_port=number],  
        [timeout=number] }
```

where:

host is the IP address or name of the remote host

port is the TCP port of the remote host

interface_addr is the IP address of the local interface to be used. If not specified, the local interface to be used is assigned by the operating system

local_port is the local TCP port to be used. If not specified, the local port to be used is assigned by the operating system

TCP communications may also be initiated remotely and be awaited (server mode), which is started by posting an event in the form:

```
evt = { class='tcp', type='listen', [interface_addr=string],
        [local_port=number] }
```

In both cases, (type='connect' and type='listen'), the connection result is returned in a pre-registered event handler for the tcp class. When in server mode, the returned event is received each time a remote host establishes a new connection. The returned event has the following form:

```
evt = { class='tcp', type='connect', host=string, port=number,
        interface_addr=string, local_port=number,
        connection=number, error=string }
```

The *error* and *connection* fields are mutually exclusive. If there is a communication error, an error message is returned in the *error* field. If, however, the connection is successfully established, the connection identifier is returned in the *connection* field.

NOTE – If there is an attempt to establish a connection with a pair *interface_addr/local_port* already in use, the connection is unsuccessful and the *error* field must have the value 'port unavailable'.

A Lua application sends data, using the TCP, by posting events of the form:

```
evt = { class='tcp', type='data', connection=number,
        value=string, [timeout=number] }
```

Similarly, a Lua application receives data transported by the TCP by handling events of the form:

```
evt = { class='tcp', type='data', connection=number,
        value=string, error=string }
```

The *error* and *value* fields are mutually exclusive. If there is a communication error, an error message is returned in the *error* field. Otherwise, the message is passed in the *value* field.

In order to close the connection, an event of the following form shall be posted:

```
evt = { class='tcp', type='disconnect', connection=number }
```

Similarly, if the remote host closes the connection or the connection is lost, an event of the following form is returned:

```
evt = { class='tcp', type='disconnect', connection=number, error=string }
```

In order to finish the acceptance for new TCP connections (server mode), a Lua application must post an event of the form:

```
evt = { class='tcp', type='unlisten', local_port=number }
```

The class dependent filter of the "tcp" class is *connection*.

9.3.2 Extended udp class:

Lua extended udp class allows for event-driven communication using UDP protocol

A Lua application sends data, using UDP protocol, by posting events of the form:

```
evt = { class='udp', type='data', (host =string | broadcast=boolean),
        port=number, [interface_addr=string], [local_port=number],
        value=string }
```

where:

host is the IP address or name of the remote host

broadcast indicates that data must be sent to all hosts in the sub-network

port is the UDP port of the remote host

interface_addr is the IP address of the local interface to be used. If not specified, the local interface to be used is assigned by the operating system

local_port is the local TCP port to be used. If not specified, the local port to be used is assigned by the operating system

In order to receive UDP data, an association (bind) shall be firstly established through posting an event in the form:

```
evt = { class='udp', type='bind', [interface_addr=string],  
        local_port=number, [max_data_length=number] }
```

where:

max_data_length specifies the maximum length in bytes for the value field in udp data reception events for the requested association.

An association result is returned in a pre-registered event handler for the class. The returned event has the form:

```
evt = { class='udp', type='bind', [interface_addr=string],  
        local_port=number, association=number, error=string }
```

The *interface_addr* field is included in the returned event if the same field was specified in the bind event posted to establish the association.

The *error* and *association* fields are mutually exclusive. If there is an error, a message is returned in the *error* field. If, however, the association is successfully established, the association identifier is returned in the *association* field.

NOTE – If there is an attempt to establish an association with a pair *interface_addr/local_port* already in use, the association is unsuccessful and the *error* field must have the value 'port unavailable'.

A Lua application receives data transported by UDP protocol by handling events of the form:

```
evt = { class='udp', type='data', host=string, port=number,  
        association=number, value=string }
```

In order to release an association, an event of the following form shall be posted:

```
evt = { class='udp', type='unbind', association=number }
```

The class dependent filter of the "udp" class is *association*.

9.3.3 The rtp class

The rtp class allows event-driven communication using the RTP.

A Lua script sends data, using the RTP, through posting events in the form:

```
evt = { class='rtp', host_addr=string, port=number,  
        value=string, timeout=number }
```

Similarly, a Lua application receives data transported by RTP handling events in the form:

```
evt = { class='rtp', host_addr=string, port=number,  
        value=string, error_msg=string }
```

The *error_msg* and *value* fields are mutually exclusive. When there is a communication error, a message is returned in the *error_msg* field. When the communication is succeeded, the message is passed in the *value* field.

NOTE 1 – A specific middleware implementation should handle issues like authentication, connection timeout/retry or whether a connection should remain open.

NOTE 2 – In the rtp class, the class dependent filter could only be *host_addr, port*.

9.3.4 The sms class

The sms class allows event-driven communication using the SMS protocol.

The sms class is an event class must be in conformance with the sms class as specified in clause 10.3.2.2 of [ITU-T H.761].

9.3.5 The sds class

The sds class allows applications to receive events from the SDS [ITU-T H.770] functionality.

sds class

event class sds provides access to the list of available service providers and their respective services. For detailed information about SDS mechanisms, refer to [ITU-T H.770].

The information acquisition process shall be performed in two steps:

- 1) A request is made calling the asynchronous `event.post()` function;
- 2) An event, to be delivered to the registered-event handlers of a Lua script, whose data field contains a set of subfields and is represented by a Lua table. The set of subfields depends on requested information.

NOTE 1 – In the sds class, the class dependent filter could only be *type*.

NOTE 2 – Any sds event may be received without any previous request. This shall happen in case of changes in the list of available service providers or on the list of the available services.

Six event types are defined by the following tables:

type = 'providers'

The table of 'providers' event type is composed of a set of vectors, each one with information related to a service provider.

Each request for a table of 'providers' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type='providers'}),
```

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
  class = 'sds',
  type = 'providers',
  serviceproviders = {
    { -- serviceproviders is indexed like a vector
      serviceProviderIdentifier = <string>, -- Unique URL, domain name
      serviceProviderName = <string>,
      serviceProviderDescription = <string>,
      serviceProvideLogoUri = <string>,
      serviceOfferSummary = {
        { -- serviceOfferSummary is indexed like a vector
          OfferType = <number>, -- type of the service

          PushAddress = <string>
          pullUrl = <string>
          webPortalUrl = <string>

        }, ...
      },
      recordVersion = <number>
    }, ...
  }
}
```

serviceType can have the following values:

- 0 - Linear TV Discovery
- 1 - Package Discovery
- 2 - Content Guide Discovery
- 3 - Service from Other Service Provider Discovery
- 1000 - Other services Discovery

type = 'linearTVServices'

The table of the 'linearTVServices' event type is used to receive all information concerning the available linear TV services.

Each request for a table of 'linearTVServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'linearTVServices'[,  
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {  
  class = 'sds',  
  type = 'linearTVServices',  
  lineartvservices = {  
    { -- lineartvservices is indexed like a vector  
      ServiceProviderIdentifier = <string>, -- Unique URL, domain name,  
etc  
  
      metadataServerUrl = <string>, -- if none value shall be nil and  
                                     -- TS shall have SI/PSI  
      PortalUrl = <string>, -- if none value shall be nil  
      purchaseInformationUrl = <string>, -- if none value shall  
be nil  
      serviceIdentifier = <string>, -- unique identifier for the  
service  
                                     -- in the service provider domain  
      OriginalNetworkId = <number>,  
      transportStreamId = <number>,  
      serviceId = <number>,  
      maxBitRate = <number>, -- Max bitrate in kbits/s  
      serviceLocations = { -- serviceLocations is indexed like a vector  
        ipMulticastAddress = <string>, -- if none value shall be nil  
        ipMulticastPort = <string>,  
        ipMulticastSource = <string>,  
        tvFec = { -- if none value shall be nil  
          fecBaseLayerMulticastAddress = <string>,  
          fecEnhancementLayerMulticastAddress = <string>,  
          maximumPacketNumberInBlocks = <number>,  
          maxFecBlockDuration = <number>,  
          maxSourceBlockLength = <number>, -- only for fecEnhancement  
          encodingSymbolSize = <number> -- only for fecEnhancement  
        },  
        unicastUrl = <string>, -- if none value shall be nil  
      }  
    }  
  }  
  audioCoding = <number>,  
}
```

```

        videoCoding = <number>,
        streamingType = <number>,    -- 0 RTP; 1 UDP; 1000 Other
        multiplexMode = <number>,    -- 0 TS; 1 Time-stamped TS; 1000 Other

        recordVersion = <number>
    }, ...
}

```

type = 'servicePackages'

The table of the 'servicePackages' event type is used to receive information of how the services are packaged together.

Each request for a table of 'servicePackages' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'servicePackages'[,
providerIdentifier] } ),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```

evt = {
  class = 'sds',
  type = 'servicePackages',
  servicePackages = {
    { -- servicePackages is indexed like a vector
      ServiceProviderIdentifier = <string>, -- Unique URL, domain name
      etc
      PackageIdentifier = <string>,
      displayable = <number>,
      packageName = <string>,
      packageDescription = <string>,
      preferredContentGuideIdentifier = <string>,
      individualservices = {
        { -- inividualservices is indexed like a
vector
          serviceProviderIdentifier = <string>, --if nil, then service
provider is the same from service package
          serviceIdentifier = <string>
        }, ...
      },
      recordVersion = <number>
    }, ...
  }
}

```

type = 'contentGuideServices'

The table of the 'contentGuideServices' event type is used to receive information about content guide services.

Each request for a table of 'contentGuideServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'contentGuideServices'[,
providerIdentifier] })),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
  class = 'sds',
  type = 'contentGuideServices',
  contentguideservices = {
    { -- contentguideservices is indexed like a vector
      ServiceProviderIdentifier = <string>, -- Unique URL, domain name
    etc
      contentGuideIdentifier = <string>,
      contentGuideName = <string>,
      contentGuideProviderName = <string>,
      contentGuideDescription = <string>,
      contentGuideLocator = <string>,
      contentGuideType = <number>, -- 0 linearTV; 1 vod; 2 both;3 other
      contentGuideLogoUri = <string>,
      targetServiceProviderIdentifier = <string>,

      recordVersion = <number>
    }, ...
  }
}
```

type = 'thirdPartyServices'

The table of the 'thirdPartyServices' event type is used to receive information about third party services.

Each request for a table of 'thirdPartyServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'thirdPartyServices'[,
providerIdentifier] })),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```
evt = {
  class = 'sds',
  type = 'thirdPartyServices',
  thirdpartyservices = {
    { -- thirdpartyservices is indexed like a vector
      ServiceProviderIdentifier = <string>, -- Unique URL, domain name
    etc
```

```

        referencedServiceProviderIdentifier = <string>, -- Unique URL,
domain name etc
        individualServices = { -- if nil, then the entire set of offerings
from the service provider is referenced
            {
                -- inividualservices is indexed like a
vector
                serviceIdentifier = <string>
            }, ...
        recordVersion = <number>
    }, ...
}

```

type = 'otherServices'

The table of the 'otherServices' event type is used to receive information about third party services.

Each request for a table of 'otherServices' event type shall be carried out through the following call:

```
event.post('out', { class='sds', type= 'otherServices'[,
providerIdentifier] }),
```

where:

i) the *providerIdentifier* field is obtained from the table *serviceproviders*. If the *providerIdentifier* is not specified, then the list of linear TV services of all available service providers shall be requested.

The returned event is created after all requested information is received and processed by the middleware (information that is not received within a maximum interval shall be returned as 'nil'). The data table is returned as follows:

```

evt = {
    class = 'sds',
    type = 'otherServices',
    otherservices = {
        {
            -- otherservices is indexed like a
vector
            ServiceProviderIdentifier = <string>, -- Unique URL, domain name
etc
            serviceIdentifier = <string>,
            serviceName = <string>,
            serviceDescription = <string>,
            serviceType = <number>, -- 0 email; 1 SMS; 2 ebook; 1000 other
            serviceLocator = <string>,
            recordVersion = <number>
        }, ...
    }
}

```

9.3.6 The smartcard class

The smartcard class allows handling of events generated by smartcard devices.

smartcard Class

```
evt = {class = "smartcard", slot:number, eventId:number }
```

slot is the number of the smartcard slot the event refers to

eventId is the type of event. It can be:

- 0 - CARD_INSERTED. Event when a card has been inserted
- 1 - CARD_MUTE. Event when a card is in mute

- 2 - CARD_NOT_PRESENT. Event when no card is present
- 3 - CARD_REMOVED. Event when a card is removed
- 4 - CARD_TERMINAL_REMOVED. Event when the card terminal is removed

9.3.7 The pvr class

The pvr class allows handling of events generated by PVR operations.

pvr Class

```
evt = { class = "pvr", eventId: number }
```

eventId is the type of event. It can be:

- 3 - PVR_ERROR_PERMISSION_DENIED
- 4 - PVR_ERROR_NOT_ENOUGH_SPACE
- 10 - PVR_ERROR_UNKONW
- 5 - PVR_EVENT_5MIN_START - indicates a recording starts in 5 minutes
- 6 - PVR_EVENT_RECORD_STOP - indicates the end of a recording operation

9.3.8 The widget class

The widget class allows handling of events generated by graphical widgets.

widget class:

The widget event class provides access to event information related to graphical widgets.

NOTE – In the widget class, the class dependent filter could only be *type*.

For any given type, the event is returned as follows:

```
evt = {
  class      = 'widget',
  type       = <string>,
  eventId    = <number>,
  sourceobj  = <object>,
  data      = {...}
}
```

The following event types are defined:

type='textinput'

The table of the 'textinput' event type contains information about an asynchronous event occurred on a textinput widget, referred by *sourceobj*. The data field depends on event subtype and can encapsulate an inner table with key modifiers, e.g. "modifiers={alt, ctrl}" specifying any modifier keys pressed by the user.

eventId identifies an event subtype, as follows:

- 01 - WIDGET_FOCUSED - data field is nil
- 02 - WIDGET_UNFOCUSED - data field is nil
- 11 - WIDGET_KEYPRESSED - data field contains key value and modifiers – 'key="9"

modifiers={META}

- 12 - WIDGET_KEYRELEASED - data field contains key value and modifiers – 'key="9"

modifiers={SHIFT}

Possible values for the "modifiers" table are: META, ALT, CTRL, SHIFT, FN

type='cursor'

The table of the 'cursor' event type contains information about an asynchronous event occurred from a cursor widget, referred by *sourceobj*. The data field depends on event subtype.

eventId identifies an event subtype, as follows:

- 01 - WIDGET_CURSOROVER - data field contains cursor coordinates – 'x=12,y=105'
- 02 - WIDGET_CURSORIN - data field contains cursor coordinates – 'x=150,y=213'

03 - WIDGET_CURSOROUT - data field contains cursor coordinates - 'x=150,y=213'
11 - WIDGET_CURSORPRESS - data field contains cursor coordinates and a button identifier -
'x=150,y=213,button=1'
12 - WIDGET_CURSORRELEASE - data field contains cursor coordinates and button identifier -
'x=150,y=213,button=1'
13 - WIDGET_CURSORDOUBLECLICK - data field contains cursor coordinates and button identifier -
'x=150,y=213,button=1'

Possible values for the button identifier are:

01 - WIDGET_LEFTBUTTON
02 - WIDGET_MIDDLEBUTTON
03 - WIDGET_RIGHTBUTTON

Annex A

Lua specification

(This annex forms an integral part of this Recommendation.)

The content of this annex was extracted from [b-Lua 5.1], and it is reprinted here with approval of the authors.

A.1 The language

A.1.1 Notation used

The language constructs are explained using the usual extended Backus–Naur form (BNF) notation, in which {a} means 0 or more instances of a, and [a] means an optional a. Keywords are shown in bold, non-terminals are shown in the standard document font, and other terminal symbols are also shown in the standard document font, but enclosed in single quotes. The complete syntax of Lua appears in clause A.7.

A.1.2 Lexical conventions

Names (also called identifiers) in Lua may be any string of letters, digits and underscores not beginning with a digit. This coincides with the definition of names in most languages. (The definition of letter depends on the current locale: any character considered alphabetic in the current locale may be used in an identifier.) Identifiers are used to name variables and table fields.

The keywords in Table A.1 are reserved and shall not be used as names.

Table A.1

and	break	do	else	elseif	end
false	for	function	if	in	local
nil	not	or	repeat	return	then
true	until	while			

Lua is a case-sensitive language: and is a reserved word, but And and AND are two different, valid names. As a convention, names starting with an underscore followed by uppercase letters (such as _VERSION) are reserved for internal global variables used by Lua.

The following strings denote other tokens:

+	-	*	/	%	^	#
==	~=	<=	>=	<	>	=
(	)	{	}	[	]	
;	:	,	.	..	...	

Literal strings may be delimited by matching single or double quotes, and may contain the following C-like escape sequences:

- \a --- bell
- \b --- backspace
- \f --- form feed
- \n --- newline
- \r --- carriage return
- \t --- horizontal tab
- \v --- vertical tab

- `\\` --- backslash
- `\"` --- quotation mark (double quote)
- `\'` --- apostrophe (single quote)

Moreover, a `\newline` (i.e., a backslash followed by a real newline) results in a newline in the string. A character in a string may also be specified by its numerical value using the escape sequence `\ddd`, where `ddd` is a sequence of up to three digits. (Note that if a numerical escape is to be followed by a digit, it must be expressed using exactly three digits.) Strings in Lua may contain any 8-bit value, including embedded zeros, which may be specified as `\0`.

To put a double (single) quote, a newline, a backslash or an embedded zero inside a literal string enclosed by double (single) quotes, an escape sequence is required. Any other character may be directly inserted into the literal. (Some control characters can cause problems for the file system, but Lua has no problem with them.)

Literal strings may also be defined using a long format enclosed by long brackets. An opening long bracket of level `n` is defined as an opening square bracket followed by `n` equal signs followed by another opening square bracket. So, an opening long bracket of level 0 is written as `[[`, an opening long bracket of level 1 is written as `[=`, and so on. A closing long bracket is defined similarly; for instance, a closing long bracket of level 4 is written as `]===]`. A long string starts with an opening long bracket of any level and ends at the first closing long bracket of the same level. Literals in this bracketed form can run for several lines, do not interpret any escape sequences, and ignore long brackets of any other level. They may contain anything except a closing bracket of the proper level or embedded zeros.

For convenience, when the opening long bracket is immediately followed by a newline, the newline is not included in the string. As an example, in a system using American standard code for information interchange (ASCII) (in which `'a'` is coded as 97, newline is coded as 10, and `'1'` is coded as 49), the four literals below denote the same string:

```
(1)      'alo\n123"'
(2)      "alo\n123\""
(3)      '\97l0\10\04923"'
(4)      [[alo
          123"]]
(5)      [==[
          alo
          123"]]=]
```

Numerical constants may be written with an optional decimal part and an optional decimal exponent. Examples of valid numerical constants are:

```
3      3.0      3.1416  314.16e-2  0.31416E1
```

Comments start with a double hyphen (`--`) anywhere outside a string. If the text immediately after `--` is not an opening long bracket, the comment is a short comment, which runs until the end of the line. Otherwise, it is a long comment, which runs until the corresponding closing long bracket. Long comments are frequently used to disable code temporarily.

A.1.3 Values and types

A.1.3.1 Basic types

Lua is a dynamically typed language. That means that variables do not have types; only values do. There are no type definitions in the language. All values carry their own type.

All values in Lua are first-class values. That means that all values may be stored in variables, passed as arguments to other functions and returned as results.

There are eight basic types in Lua: nil, boolean, number, string, function, userdata, thread and table. Nil is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value. Boolean is the type of the values: false or true. Both nil and false make a condition false; any other value makes it true. Number represents real (double-precision floating-point) numbers. (It is easy to build Lua interpreters that use other internal representations for numbers, such as single-precision float or long integers. See file luaconf.h [b-luaconf.h].) String represents arrays of characters. Lua is 8-bit clean: Strings may contain any 8-bit character, including embedded zeros ('\0') (see clause A.1.2).

Lua may call (and manipulate) functions written in Lua and functions written in C (see clause A.1.6.9).

The type userdata is provided to allow arbitrary C data to be stored in Lua variables. This type corresponds to a block of raw memory and has no pre-defined operations in Lua, except assignment and identity test. However, by using metatables, the programmer may define operations for userdata values (see clause A.1.9). Userdata values shall not be created or modified in Lua, only through the C API. This guarantees the integrity of data owned by the host program.

The type thread represents independent threads of execution and it is used to implement coroutines (see clause A.1.12). Do not confuse Lua threads with operating-system threads. Lua supports coroutines on all systems, even those that do not support threads.

The type table implements associative arrays, i.e., arrays that may be indexed not only with numbers, but also with any value (except nil). Tables may be heterogeneous; i.e., they may contain values of all types (except nil). Tables are the sole data structuring mechanism in Lua; they may be used to represent ordinary arrays, symbol tables, sets, records, graphs, trees, etc. To represent records, Lua uses the field name as an index. The language supports this representation by providing a.name as syntactic sugar for a["name"]. There are several convenient ways to create tables in Lua (see clause A.1.6.8).

Like indices, the value of a table field may be of any type (except nil). In particular, because functions are first-class values, table fields may contain functions. Thus tables may also carry methods (see clause A.1.6.10).

Strings, tables, functions and userdata values are objects: variables do not actually contain these values, only references to them. Assignment, parameter passing and function returns always manipulate references to such values; these operations do not imply any kind of copy.

The library function type returns a string describing the type of a given value.

A.1.3.2 Coercion

Lua provides automatic conversion between string and number values at run time. Any arithmetic operation applied to a string tries to convert that string to a number, following the usual conversion rules. Conversely, whenever a number is used where a string is expected, the number is converted to a string, in a reasonable format. For complete control over how numbers are converted to strings, use the format function from the string library (see string.format).

A.1.4 Variables

Variables are places that store values. There are three kinds of variables in Lua: global variables, local variables, and table fields.

A single name may denote a global variable or a local variable (or a function formal parameter, which is a particular kind of local variable):

```
var ::= Name
```

Name denotes identifiers, as defined in clause A.1.2.