tclvisa Version 0.3.0
Programmer's Manual

July 25, 2012

Contents

Chapter 1

Getting Started

1.1 Introduction

`tclvisa` is a standard Tcl extension providing a binding to Virtual Instrument Software Architecture (VISA) API. It allows to use VISA functionality from regular Tcl scripts via set of commands. Most of the commands have similar names and are intuitive to use for the one who knows VISA API. For example, Tcl command `visa::open` is a front-end for `viOpen` VISA function. The specifications of VISA can be found here:

http://www.ivifoundation.org/specifications/default.aspx.

As an extention `tclvisa` follows conventions of Tcl Extension Architecture (TEA). It is loaded dynamically into Tcl shell or Tcl-based application on demand. Please refer to TEA documentations for details:

http://www.tcl.tk/doc/tea/.

1.2 Installation

Prior to usage of `tclvisa` one need VISA implementation installed. There are several known implementations (e.g. National Instruments VISA, or Agilent IO Library Suite). VISA libraries should be installed in proper directories and be available for linking.

`tclvisa` itself should be installed as a typical Tcl extension. Partcular details of installation, such as target directory, are platform-dependent. Please refer to the documentation of your Tcl version.

If you have no access to system directories to install `tclvisa` into, you can install into arbitrary directory. In order to make Tcl known about this directory, you should add path to it to `TCLLIBPATH` environment variable.

1.2.1 Installation in Microsoft Windows

Binary instalation package for Microsoft Windows can be downloaded from `tclvisa` site. This package contains DLL with compiled library code, documentation and demo scripts. All downloads can be found here:

http://sourceforge.net/projects/tclvisa/files/.

1.3 Use in Tcl

In order to start using `tclvisa` within Tcl one should issue following command:

```
package require tclvisa
```

If the library is properly installed, this command returns version of `tclvisa` loaded. Library is ready to use. All commands and predefined variables are placed in `visa::` namespace.

Chapter 2

Programming with `tclvisa`

2.1 VISA Constants

VISA defines a lot of predefined contants with codes of attributes, errors etc. Most of these constants are available in Tcl within `visa` namespace and without `VI_` prefix.

For example, `VI_EXCLUSIVE_LOCK` constant is represented by `visa::EXCLUSIVE_LOCK` variable in Tcl. When using these predefined variables in expressions, do not forget adding `$` prefix prior to variable name.

Following example demonstrates usage of predefined constant:

```
# open instrument exclusively
set vi [visa::open $rm "ASRL2::INSTR" $visa::EXCLUSIVE_LOCK]
```

2.2 Channels

Most of VISA functions operate with *sessions* which are represented in C language by `viSession` type. In Tcl these sessions are stored in standard channels. E. g. `visa::open` returns name of Tcl channel to be used in subsequent operations. This approach has following benefits:

- Standard IO procedures are used in VISA IO: `puts`, `gets` etc.
- VISA sessions may be transparently passed to any procedure which accepts Tcl channel. If some third-party library works with regular Tcl channels, it therefore can read/write from/to VISA device.
- Tcl automatically closes all opened channels when interpeter terminates.

`tclvisa` provides a procedure to open VISA sessions: `visa::open`, which is a front-end of `viOpen` VISA function. But `tclvisa` does not provide front-ends for `viClose` or `viWrite` VISA functions. Instead one should use standard Tcl commands, such as `close` or `puts`. Tcl detects type of the channel and calls proper VISA function internally. For example, when `close` is issued on VISA channel, opened by `visa::open`, then `viClose` function is actually called within Tcl internals.

Following example demonstrates usage of VISA channel:

```
# open instrument
set vi [visa::open $rm "ASRL2::INSTR"]

# send "reset" command to instrument
puts $vi "*RST"

# close VISA session
close $vi
```

See also table of correspondence between supported VISA functions and Tcl commands.

2.2.1 Buffering

VISA IO functions, such as `viWrite` or `viRead`, by default work with *messages*, where message is a sequence of bytes of arbitrary length followed by special "end-of-message" character. For example, SCPI messages end with "new line" character (ASCII code is `0Ah`). When I send a message terminated by "end-of-message" via `viWrite`, I can be sure that it is actually sent to the device rather than kept in intermediate buffer. When I call `viRead`, it returns immediately after receiving of "end-of-message" character regardless of the length of input buffer.

Tcl channels by default work with continuous streams of bytes. IO functions typically block until IO buffer is full or "end-of-data" is detected. For example, `read` command called without a buffer length specified on file channel blocks until entire file is read. It's evident that this approach does not work with message-based protocols like SCPI.

Fortunately Tcl offers different buffering options, which can be set or red by `fconfigure` command. One of them is "`-buffering line`" which tells Tcl finish current IO operation when "end-of-line" character is received or sent. When buffering type is "`line`", `read` blocks until "end-of-line" is received, and `puts` actually sends data right after "end-of-line" is found in

outcoming data. In the terms or SCPI commands, `read` blocks until complete response is received from a device.

When VISA channel is created by `visa::open`, buffering type is automatically set to "`line`". If one needs to switch channel mode, then `fconfigure` command with proper `-buffering` option should be issued.

2.2.2 IO Timeouts

In VISA API IO message communication timeouts can be specified or read by `viSetAttribute` and `viGetAttribute` functions where `attribute` parameter is set to `VI_ATTR_TMO_VALUE`.

In `tclvisa` timeout can be controlled in similar way via `visa::set-attribute` and `visa::get-attribute` commands. But preferred and more laconic approach is to use `fconfigure` Tcl command with standard `-timeout` option. Look at the example below:

```
# open an instrument
set vi [visa::open $rm "ASRL1::INSTR"]

# read current timeout value
set tm [fconfigure $vi -timeout]

# set new timeout value
fconfigure $vi -timeout [expr 2 * $tm]
```

In this example we read current timeout value, then set a new value that is twice the original one.

Inside the `tclvisa` these invocations of `fconfigure` are converted to corresponding calls of `viGetAttribute` and `viSetAttribute` VISA API functions.

2.2.3 Non-blocking IO

Standard Tcl channels have a `-blocking` option which "determines whether I/O operations on the channel can cause the process to block indefinitely" (quote from the `fconfigure` manual).

VISA API does not support non-blocking IO natively, probably because it offers asynchronous operations for that. In `tclvisa` non-blocking IO is *emulated* by setting IO timeout to zero.

I. e. when user sets a VISA channel to non-blocking mode by `fconfigure` command with `-blocking 0` option, `tclvisa` internally sets IO timeout for

this channel to zero. When channel is reverted back to the blocking mode (that is the default state for all VISA channels), timeout is restored to the previous value.

See also "Suppressed Errors" section on page 10.

2.2.4 Anynchronous IO

Tcl `fileevent` command cannot be called upon VISA channel. This functionality is not implemented yet.

Support of `viWriteAsync` and `viReadAsync` VISA API functions is not implemented too.

2.2.5 Serial-Specific Options

When a standard Tcl channel is backed by a serial port, it has a set of specific options that control baud speed, parity etc.

VISA instruments which are connected to the serial port (their addresses start with `ASRL` prefix) have full set of corresponding attributes, such as `VI_ATTR_ASRL_BAUD`, `VI_ATTR_ASRL_PARITY` and similar. In order to configure, for instance, baud rate one can use `visa::set-attribute` command with `$visa::ATTR_ASRL_BAUD` passed as an attribute name. But preferred way is to use `fconfigure` Tcl command and standard options. See example:

```
# open an instrument
set vi [visa::open $rm "ASRL1::INSTR"]

# set baud rate, parity, word length and stop bits
fconfigure $vi -mode 9600,n,8,1
```

From the Tcl code's point of view channel '`vi`' behaves like a regular serial port. For example, this channel can be transparently passed to third-party library that implements some serial-based protocol.

All serial-specific options supported and corresponding VISA attributes are listed in fig. 2.1. Format of each option is described in the `fconfigure` command manual.

See also `demo/fconfigure.tcl` script that demonstrates usage of channel options.

2.3 Error Handling

All VISA API functions returns `viStatus` value with error code. Zero code means successfull completion, positive value means that operation returns

Figure 2.1: Serial-specific channel options and VISA equivalents.

Tcl option	VISA Attribute(s)
handshake	VI_ATTR_ASRL_FLOW_CNTRL
mode	VI_ATTR_ASRL_BAUD, VI_ATTR_ASRL_PARITY, VI_ATTR_ASRL_DATA_BITS, VI_ATTR_ASRL_STOP_BITS
queue	VI_ATTR_ASRL_AVAIL_NUM
ttycontrol	VI_ATTR_ASRL_DTR_STATE, VI_ATTR_ASRL_RTS_STATE, VI_ATTR_ASRL_BREAK_STATE
ttystatus	VI_ATTR_ASRL_CTS_STATE, VI_ATTR_ASRL_DSR_STATE, VI_ATTR_ASRL_RI_STATE, VI_ATTR_ASRL_DCD_STATE
xchar	VI_ATTR_ASRL_XON_CHAR, VI_ATTR_ASRL_XOFF_CHAR

some warning which may be ignored in the most of cases. Negative code means error that should be handled by application.

In scripting language like Tcl the API developer should follow "KISS" principle and make things as simple as possible. This is why `tclvisa` commands do not explicitly return error code to the calling script. Instead they return either actual result of operation (say, instance of the new channel created by `visa::open` command) or nothing, when operation has no any meaningful result (e. g. `visa::set-attribute`).

When underlying VISA API function returns error, and `tclvisa` cannot handle this error itself, it throws an exception which can be handled by `catch` command in the calling script. This is a standard and expected behaviour for Tcl command. For example, standard `open` command throws exception when it cannot open a file. Exception handler receives a string with error code and description in the following format:

```
[CODE] Text description
```

where `CODE` is the name of the predefined VISA error constant.

In the following example we're trying to open an instrument that does not actually exist:

```
# this attempt should return VI_ERROR_RSRC_NFOUND error
if { [catch { set vi [visa::open $rm ASRL99::INSTR] } rc] } {
  puts "Error: $rc"
}
```

This code produces following output:

```
Error: [VI_ERROR_RSRC_NFOUND] Insufficient location information
 or resource not present in the system.
```

Calling side then can parse the error string to retrieve VISA error code which is placed between square brackets.

2.3.1 Suppressed Errors

Some errors returned from VISA functions are suppressed by `tclvisa` , i. e. do not cause Tcl exceptions:

- Timeout error (`VI_ERROR_TMO`) is suppressed by IO operations on VISA channels. For example, when read attempt expires, the `read` command simply returns empty string.
- `VI_ERROR_RSRC_NFOUND` error returned by `viFindRsrc` or `viFindNext` is suppressed by `visa::find` command. For example, when search criteria do not produce any result, this command simply returns an empty list.
- `VI_ERROR_INV_RSRC_NAME` and `VI_ERROR_RSRC_NFOUND` errors returned by `viParseRsrc` are suppressed by `visa::parse-rsrc` command.

One can use `visa::last-error` command to determine the exact status of the last VISA operation. This command returns all errors, including suppressed ones.

2.4 Correspondence Between VISA Functions and Tcl Commands

Table on the figure 2.2 contains list of VISA API functions supported by `tclvisa` and corresponding commands to use in Tcl.

Figure 2.2: VISA functions and Tcl equivalents.

VISA API function	Tcl Command
viAssertIntrSignal	visa::assert-intr-signal
viAssertTrigger	visa::assert-trigger
viAssertUtilSignal	visa::assert-util-signal
viClear	visa::clear
viClose	close
viFindNext, viFindRsrc	visa::find
viGetAttribute	visa::get-attribute
viGpibCommand	visa::gpib-command
viGpibControlATN	visa::gpib-control-atn
viGpibControlREN	visa::gpib-control-ren
viGpibPassControl	visa::gpib-pass-control
viGpibSendIFC	visa::gpib-send-ifc
viLock	visa::lock
viOpen	visa::open
viOpenDefaultRM	visa::open-default-rm
viParseRsrc	visa::parse-rsrc
viPrintf	format, puts
viQueryf	format, puts, gets, scan
viRead	read
viReadToFile	visa::read-to-file
viScanf	gets, scan
viSetAttribute	visa::set-attribute
viUnlock	visa::unlock
viWrite	puts
viWriteFromFile	visa::write-from-file

Chapter 3

`tclvisa` Command Reference

3.1 visa::assert-intr-signal

Purpose

Asserts the specified interrupt or signal. This command is a front-end for `viAssertIntrSignal` VISA API function.

Syntax

```
visa::assert-intr-signal session mode ?statusID?
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `mode` — This specifies how to assert the interrupt. Valid value is one of the predefined `visa::ASSERT_xxx` constants. Please refer to your VISA documentation for detailed help.
- `statusID` — This is the status value to be presented during an interrupt acknowledge cycle. This argument may be omitted on certail bus types.

Return Value

None

Example

```
# open instrument
set vi [visa::open $rm "ASRL1::INSTR"]

# assert signal
visa::assert-intr-signal $vi $visa::ASSERT_USE_ASSIGNED
```

See Also

`visa::assert-util-signal`

3.2 visa::assert-trigger

Purpose

Asserts software or hardware trigger. This command is a front-end for `viAssertTrigger` VISA API function.

Syntax

```
visa::assert-trigger session protocol
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `protocol` — Trigger protocol to use during assertion. Valid value is one of the predefined `visa::TRIG_PROT_xxx` constants. Please refer to your VISA documentation for detailed help.

Return Value

None

Example

```
# open instrument
set vi [visa::open $rm "ASRL1::INSTR"]

# assert trigger
```

```
visa::assert-trigger $vi $visa::TRIG_PROT_DEFAULT
```

3.3 visa::assert-util-signal

Purpose

Asserts or deasserts the specified utility bus signal. This command is a front-end for `viAssertUtilSignal` VISA API function.

Syntax

```
visa::assert-util-signal session line
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `line` — Specifies the utility bus signal to assert. Valid value is one of the predefined `visa::UTIL_xxx` constants. Please refer to your VISA documentation for detailed help.

Return Value

None

Example

```
# open instrument
set vi [visa::open $rm "ASRL1::INSTR"]

# assert signal
visa::assert-util-signal $vi $visa::UTIL_ASSERT_SYSRESET
```

See Also

`visa::assert-intr-signal`

3.4 visa::clear

Purpose

Clears a device. This command is a front-end for `viClear` VISA API function.

Syntax

```
visa::clear session
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.

Return Value

None

Example

```
# open instrument with default access mode and timeout
set vi [visa::open $rm "ASRL1::INSTR"]

# set device to known state
visa::clear $vi
```

See Also

`visa::open`

3.5 visa::find

Purpose

Queries a VISA system to locate the resources associated with a specified interface. This command is a front-end for `viFindRsrc` and `viFindNext` VISA API functions.

Syntax

```
visa::open RMsession expr
```

Arguments

- `RMsession` — channel containing reference to open Resource Manager session opened by `visa::open-default-rm`.
- `expr` — regular expression followed by an optional logical expression. Refer to VISA API documentation for detailed discussion.

Return Value

Tcl list with addresses of all resources found. If no resources found that match the given expression, empty list is returned.

Example

```
# open resource manager session
set rm [visa::open-default-rm]

# get addresses of all serial instruments
foreach addr [visa::find $rm "ASRL?*INSTR"] {
  # address is in $addr variable
}
```

See Also

`visa::open-default-rm`

3.6 visa::get-attribute

Purpose

Retrieves the state of an attribute. This command is a front-end for `viGetAttribute` VISA API function.

Syntax

```
visa::get-attribute session attribute
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `attribute` — Integer value with ID of the VISA attribute to retrieve. Use one of the predefned `visa::ATTR_XXX` constants.

Return Value

Attribute value.

Example

```
# open instrument with default access mode and timeout
set vi [visa::open $rm "ASRL1::INSTR"]

# retrieve current baud rate of a serial bus
set baud [visa::get-attribute $vi $visa::ATTR_ASRL_BAUD]
```

See Also

`visa::set-attribute`

3.7 visa::gpib-command

Purpose

Write GPIB command bytes on the bus. This command is a front-end for `viGpibCommand` VISA API function.

Syntax

```
visa::gpib-command session buf ?count?
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `buf` — String containing valid GPIB commands.

- `count` — Number of bytes to be written. If argument is omitted, string length of `buf` is assumed.

Return Value

Number of bytes actually transferred.

Example

```
# send command
set cmd ... # this variable contains command
set ret [visa::gpib-command $vi $cmd 10]
puts "$ret bytes are transmitted"
```

3.8 visa::gpib-control-atn

Purpose

Specifies the state of the ATN line and the local active controller state. This command is a front-end for `viGpibControlATN` VISA API function.

Syntax

```
visa::gpib-control-atn session mode
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `mode` — Specifies the state of the ATN line and optionally the local active controller state. Valid value is one of the `visa::GPIB_ATN_xxx` predefined constants. Please refer to your VISA documentation for detailed help.

Return Value

None

Example

```
# open a GPIB interface device
set vi [visa::open ...
# set "assert" state
visa::gpib-control-atn $vi $visa::GPIB_ATN_ASSERT
```

See Also

`visa::gpib-control-ren`

3.9 visa::gpib-control-ren

Purpose

Controls the state of the GPIB Remote Enable (REN) interface line, and optionally the remote/local state of the device. This command is a front-end for `viGpibControlREN` VISA API function.

Syntax

```
visa::gpib-control-ren session mode
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `mode` — Specifies the state of the REN line and optionally the device remote/local state. Valid value is one of the `visa::GPIB_REN_xxx` predefined constants. Please refer to your VISA documentation for detailed help.

Return Value

None

Example

```
# open a GPIB interface device
set vi [visa::open ...
# set "assert" state
```

```
visa::gpib-control-ren $vi $visa::GPIB_REN_ASSERT
```

See Also

`visa::gpib-control-atn`

3.10 visa::gpib-pass-control

Purpose

Tell the GPIB device at the specified address to become controller in charge (CIC). This command is a front-end for `viGpibPassControl` VISA API function.

Syntax

```
visa::gpib-pass-control session primAddr ?secAddr?
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `primAddr` — Primary address of the GPIB device to which you want to pass control.
- `secAddr` — Secondary address of the targeted GPIB device. If argument is omitted, default `VI_NO_SEC_ADDR` value is assumed.

Return Value

None

Example

```
# open a GPIB device
set vi [visa::open ...
# affect the device at primary address 1
#   and without secondary address
visa::gpib-pass-control $vi 1
```

3.11 visa::gpib-send-ifc

Purpose

Pulse the interface clear line (IFC) for at least 100 microseconds. This command is a front-end for `viGpibSendIFC` VISA API function.

Syntax

```
visa::gpib-send-ifc session
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.

Return Value

None

Example

```
# open a GPIB device
set vi [visa::open ...
# send a signal
visa::gpib-send-ifc $vi
```

3.12 visa::last-error

Purpose

Returns last error occured on the channel or Resource Manager session. This command has no VISA API equivalent.

Syntax

```
visa::last-error session
```

Arguments

- `session` — channel containing reference to an either resource session opened by `visa::open` or Resource Manager session opened by `visa::open-default-rm`.

Return Value

List of three elements:

- Numeric value of the last error.
- Character code of the last error that refers to a name of the corresponding C language macro. For example, "`VI_ERROR_TMO`".
- Textual description of the last error or empty value if no error.

Notes

This command is especially useful when IO operations fail, because exact VISA error is not translated to client code by standard Tcl IO procedures, such as `puts` or `read`. In other words, when IO procedure (say, `puts`) fails on a `tclvisa` channel, only way to know what exactly occured is to call `visa::last-error`.

Only result of last operation is stored. All subsequent calls of `tclvisa` or IO commands on a channel rewrite error information.

The Resource Manager session holds result of last operation the session is used in, for example `visa::open` or `visa::find`.

Example

In the following example we're reading from an instrument and checking whether it timed out.

```
# read from device
set ans [gets $vi]

if { $ans == "" } {
  # Either timeout error or empty device response
  set err [visa::last-error $vi]
  if { [lindex $err 0] == $visa::ERROR_TMO } {
    puts stderr "Error [lindex $err 1] reading from a device"
    puts stderr "[lindex $err 2]"
```

```
  }
}
```

If the read operation timed out, following message will be displayed:

```
Error VI_ERROR_TMO reading from a device
The read/write operation was aborted because timeout expired
while operation was in progress.
```

3.13 visa::lock

Purpose

Establishes an access mode to the specified resource. This command is a front-end for `viLock` VISA API function.

Syntax

```
visa::lock session ?lockType?  ?timeout?  ?requestedKey?
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `lockType` — integer value determining type of locking. May be either `visa::EXCLUSIVE_LOCK` or `visa::SHARED_LOCK`. If argument is omitted, `visa::EXCLUSIVE_LOCK` is assumed.
- `timeout` — timeout of getting lock. If argument is omitted, infinite timeout is assumed.
- `requestedKey` — name of the shared lock to acquire. If exclusive locking is requested, this argument is ignored.

Return Value

- If an exclusive lock is requiested and acquired, procedure returns nothing.
- If an shared lock is requiested and acquired, procedure returns name of the lock.

Example

```
# get exclusive lock and wait forever
visa::lock $vi

# get exclusive lock and wait 5 seconds
visa::lock $vi $visa::EXCLUSIVE_LOCK 5000

# get shared lock and wait 5 seconds
set key [visa::lock $vi $visa::SHARED_LOCK 5000 "MYLOCK"]
```

See Also

`visa::open`, `visa::unlock`

3.14 visa::open

Purpose

Opens a session to the specified resource. This command is a front-end for `viOpen` VISA API function.

Syntax

```
visa::open RMsession rsrcName ?accessMode?  ?openTimeout?
```

Arguments

- `RMsession` — channel containing reference to open Resource Manager session opened by `visa::open-default-rm`.
- `rsrcName` — name of the VISA resource to connect to.
- `accessMode` — integer parameter determining access mode. May be bitwise OR combination of the following constants:
  - `visa::EXCLUSIVE_LOCK` — acquire exclusive lock to a resource;
  - `visa::LOAD_CONFIG` — use external configuration;

  Refer to VISA documentation for more details about access mode. If parameter is omitted, default zero value is used.

- `openTimeout` — operation timeout. If parameter is omitted, default timeout value is used.

Return Value

Tcl channel with reference to opened VISA session. This channel can be used in standard Tcl IO procedures, like `puts`.

Notes

There is no a Tcl wrapper for `viClose` VISA API function. In order to close a VISA session one should use standard Tcl `close` command instead, which calls `viClose` internally.

Example

```
# open resource manager session
set rm [visa::open-default-rm]

# open instrument with default access mode and timeout
set vi1 [visa::open $rm "ASRL1::INSTR"]

# open instrument exclusively
set vi2 [visa::open $rm "ASRL2::INSTR" $visa::EXCLUSIVE_LOCK]
```

See Also

`visa::open-default-rm`

3.15 visa::open-default-rm

Purpose

Returns a session to the Default Resource Manager resource. This command is a front-end for `viOpenDefaultRM` VISA API function.

Syntax

`visa::open-default-rm`

Arguments

None

Return Value

Tcl channel with reference to opened resource manager session. This channel can be used in subsequent `tclvisa` procedure calls.

Notes

There is no a Tcl wrapper for `viClose` VISA API function. In order to close a VISA session one should use standard Tcl `close` command instead, which calls `viClose` internally.

Example

```
# open resource manager session
set rm [visa::open-default-rm]

# use session reference
...

# close session
close $rm
```

See Also

`visa::open`

3.16 visa::set-attribute

Purpose

Sets the state of an attribute. This command is a front-end for `viSetAttribute` VISA API function.

Syntax

```
visa::set-attribute session attribute attrState
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `attribute` — Integer value with ID of the VISA attribute to set. Use one of the predefned `visa::ATTR_XXX` constants.
- `attrState` — Integer value with desired attribute state.

Return Value

None

Example

```
# open instrument with default access mode and timeout
set vi [visa::open $rm "ASRL1::INSTR"]

# set new baud rate of a serial bus
visa::set-attribute $vi $visa::ATTR_ASRL_BAUD 19200
```

See Also

`visa::get-attribute`

3.17 visa::parse-rsrc

Purpose

Parse a resource string to get the interface information. This command is a front-end for `viParseRsrc` VISA API function.

Syntax

```
visa::parse-rsrc RMsession rsrcName
```

Arguments

- `RMsession` — channel containing reference to open Resource Manager session opened by `visa::open-default-rm`.
- `rsrcName` — Unique symbolic name of a resource.

Return Value

- On success returns Tcl list of two integers: interface type code and interface number.
- If address given is not valid or device does not exists, returns empty value.

Example

```
# open resource manager session
set rm [visa::open-default-rm]

# parse instrument address
lassign [visa::parse-rsrc $rm "ASRL1::INSTR"] intfType intfNum

if { $intfType == $visa::INTF_ASRL } {
  puts "Have serial interface device with interface number $intfNum"
}
```

See Also

`visa::open-default-rm`

3.18 visa::read-to-file

Purpose

Read data synchronously, and store the transferred data in a file. This command is a front-end for `viReadToFile` VISA API function.

Syntax

```
visa::read-to-file session fileName count
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `fileName` — name of file to which data will be written.
- `count` — number of bytes to be read.

Return Value

Number of bytes actually transferred.

Example

```
# open instrument with default access mode and timeout
set vi [visa::open $rm "ASRL1::INSTR"]

# read up to 1024 bytes of data
# or until term char is received
visa::read-to-file $vi "raw.dat" 1024
```

See Also

`visa::write-from-file`

3.19 visa::unlock

Purpose

Relinquishes a lock for the specified resource. This command is a front-end for `viUnlock` VISA API function.

Syntax

```
visa::unlock session
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.

Return Value

None

Example

```
# get exclusive lock and wait forever
visa::lock $vi
```

```
# release the lock
visa::unlock $vi
```

See Also

`visa::open`, `visa::lock`

3.20 visa::write-from-file

Purpose

Take data from a file and write it out synchronously. This command is a front-end for `viWriteFromFile` VISA API function.

Syntax

```
visa::write-from-file session fileName ?count?
```

Arguments

- `session` — channel containing reference to a VISA resource session opened by `visa::open`.
- `fileName` — name of file from which data will be read.
- `count` — number of bytes to be written. If omitted, entire file is written.

Return Value

Number of bytes actually transferred.

Example

```
# open instrument with default access mode and timeout
set vi [visa::open $rm "ASRL1::INSTR"]

# write entire file content to device
visa::write-from-file $vi "raw.dat"
```

See Also

`visa::read-to-file`