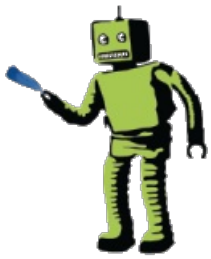
AndroWish

Executive Summary

Tcl (Tool Command Language) is a very powerful but easy to learn dynamic programming language, suitable for a very wide range of uses, including web and desktop applications, networking, administration, testing and many more. Open source and business-friendly, Tcl is a mature yet evolving language that is truly cross platform, easily deployed and highly extensible.

Tk is a graphical user interface toolkit that takes developing desktop applications to a higher level than conventional approaches. Tk is the standard GUI not only for Tcl, but for many other dynamic languages, and can produce rich, native applications that run unchanged across Windows, Mac OS X, Linux and more.



AndroWish allows to run desktop Tcl and Tk programs almost unaltered on the Android Platform while it opens the door to script a rich feature set of a mobile platform. Its sibling undroidwish uses the same code base and offers a similar feature set on various desktop and embedded platforms.

Quick Links

Documentation	Tcl/Tk 8.6 manual, ble, borg, dmtx, modbus, muzic, rfcomm, sdltk, snap7, tclcan, topcua, usbserial, uvc, wmf, v4l2, zbar, zipfs, list of extensions, list of releases, environment, undroidwish, undroidwish command line switches PDF booklet for eBook readers, excerpt of this wiki
Development	AndroWish SDK, building#1, building#2, building#3, examples
Downloads	AndroWish-debug.apk, AWSDK.zip
Experimental	Updated more frequently when the LUCK binaries are rebuilt. AndroWish-debug.apk (Architectures armeabi and x86) AndroWish64-debug.apk (Architectures armeabi-v7a, arm64-v8a, and x86_64)

Features

- Native Tcl/Tk 8.6 port for Android (version 2.3.3 or above) available for ARM and x86 processors.
- Top goal: execute existing Tcl/Tk scripts on Android without modification.
- Based on Tim Baker's earlier SDLTk project.
- X11 emulation based on AGG (Anti-Grain-Geometry) and SDL 2.0.
- Provides anti-aliased rendering of lines, circles, arcs.
- Font rendering using freetype font engine. Starting with the "Back to the Future (2015-10-21)" release, Unicode 8.0 is fully supported and Emojis can be displayed (and input with the on-screen keyboard on newer Android devices).
- Includes the 3D canvas widget which uses an OpenGL to OpenGLES 1.1 emulation for drawing on the Android platform.
- Includes the tkpath widget, an enhanced canvas with SVG like capabilities, anti-aliased rendering, alpha channel, and TrueType outline fonts.
- Mounts its constituting APK (Android package) using a built-in ZIP virtual file system as a memory mapped file.
- "Batteries Included" packaging like TclKits, i.e. many ready-to-use Tcl extensions are already bundled.
- Some Android specific facilities are exposed through SDL and usable with the sdltk command.
- Tcl commands are available to use even more Android specific facilities: borg command, ble command, rfcomm command, usbserial command.
- Some Android specific things are exposed through Environment Variables.
- A MIDI sound package is built in and described in Muzic MIDI sound package.
- An experimental rendering mode allows to use it with VR headsets like Google's cardboard.
- Many example scripts are built into the AndroWish package.
- Building AndroWish requires the Android SDK and Android NDK. A detailed description by Harald Oehlmann is available in Build custom Androwish.
- A new approach of bundling Tcl scripts with the AndroWish infrastructure is described in AndroWish SDK.
- Testing and debugging Tcl scripts on an Android device can be carried out from a development system using tkconclient. Files can be transferred using a SSH/SFTP connection as described in tkconclient. More tips can be found in Test and debug strategies on AndroWish.
- There are certain Limitations of AndroWish.
- Support for generating bar codes using ZINT and decoding bar codes using the ZBar bar code reader, and data matrix codes using the libdmtx library. See description of zbar command and dmtx command.
- Beneath and Beyond AndroWish are components in the source tree which can be recompiled for other platforms

like the Raspberry Pi and even Windows.
- Slides (PDF) from EuroTcl2014.
- Slides (PDF) from Tcl2014.
- Slides (PDF) from EuroTcl2015.
- Slides (PDF) from EuroTcl2016.
- Slides (PDF) from EuroTcl2018.
- Slides (PDF) from EuroTcl2019.

The current AndroWish-debug.apk can be downloaded here (about 36 MByte, requires "install from unknown sources" in Android settings). Prehistoric versions are still available here.

Android facilities

borg command

Name

borg - control and interact with the Android OS.

Synopsis

```
package require Borg
borg cmd ?arg ...?
```

Description

This command integrates the capabilities of Tcl/Tk with Android by way of several subcommands. These allow Tcl/Tk to go where it has never gone before by querying and controlling Bluetooth functionality, OS notifications (including device vibration and even speech), location information, etc.

Bluetooth-Related Commands

`borg bluetooth devices`

> Returns a list suited for `array set` or `dict create` commands containing the Bluetooth address and friendly name of all paired Bluetooth devices.

`borg bluetooth state`

> Returns the current Bluetooth state: `off`, `on`, `turning_off`, or `turning_on`.

`borg bluetooth scanmode`

> Returns the current Bluetooth scan mode: `connectable`, `off`, `passive`, or `visible`.

`borg bluetooth myaddress`

> Returns the Bluetooth address of the local default Bluetooth adapter.

`borg bluetooth remoteaddress` *`address`*

> Returns the friendly name for the given Bluetooth *`address`*.

`borg bluetooth on`

> Quote from Android documentation: do not use without explicit user action to turn on Bluetooth. Tries to turn the Bluetooth adapter on. Returns 1 if the adapter is already or going to be turned on, 0 otherwise.

`borg bluetooth off`

> Quote from Android documentation: do not use without explicit user action to turn off Bluetooth. Tries to turn the Bluetooth adapter off. Returns 1 if the adapter is already or going to be turned off, 0 otherwise.

For communication over Bluetooth see the description of the rfcomm command. For handling of Bluetooth Low Energy (Bluetooth Smart) devices see the description of the ble command.

USB-Related Commands

`borg usbdevices ?`*`extended`*`?`

> If *`extended`* is omitted or false, a list suitable for the `array set` or `dict create` commands containing the USB device name and vendor/product identifier of all currently connected USB devices is returned. Otherwise, i.e. *`extended`* is true, three elements per USB device are returned: USB device name, product/vendor identifier, and USB interface information as in `udev`.

`borg usbpermission` *`devname`* `?`*`ask`*`?`

> Queries permission for the USB device *`devname`* and returns 1 if the device is usable, 0 if not, and a negative number on error. If the optional boolean argument *`ask`* is specified as true, a system dialog is shown allowing the user to grant or deny permission for the USB device.

For communication over USB serial converters see the description of the usbserial command.

Network-Related Commands

`borg networkinfo`

Returns the current state of the network: `none`, `wifi`, `mobile gsm`, etc. An update of this information is indicated by the `<<NetworkInfo>>` virtual event sent to all toplevel widgets.

`borg tetherinfo`

Returns the current state of tethering as a list suited for `array set` or `dict create` with zero or three entries `active`, `available`, and `error` which usually contain interface names. An update of this information is indicated by the `<<TetherInfo>>` virtual event sent to all toplevel widgets.

Desktop-Related Commands

`borg shortcut add` *`name-of-shortcut script-to-run ?png-icon-as-base-64-string?`*

Adds an icon to the desktop, with the label *`name-of-shortcut`* specified. The script, specified as *`script-to-run`* must use an absolute path as a `file://` URI and must be readable by the user id under which the AndroWish package has been registered by the Android installer. The last (optional) parameter *`png-icon-as-base-64-string`* allows the icon graphic to be specified. If not provided, a default AndroWish icon is used. According to the guidelines on iconography icons should have an aspect ratio of 48 by 48 pixels (192 x 192 is recommended, at 4 times 48x48 pixels). Example (pseudo code):

```
package require base64
proc read_binary_file {name} { # whatever is needed to read bytes ... }
set icondata [read_binary_file "/mnt/sdcard/appicon_48_48.png"]
set iconbase64 [::base64::encode -maxlen 0 $icondata]
borg shortcut add "My App" file://mnt/sdcard/speaktest.tcl $iconbase64
```

`borg shortcut delete` *`name-of-shortcut`*

Deletes an icon from desktop (depends on Android launcher support).

Notification-Related Commands

`borg notification add` *`id title ?text icon action uri type categories component arguments?`*

Adds a notification with *`title`* and *`text`* into the Android notification area. The integer *`id`*, specified by the caller, is used to identify the notification for later modification or deletion. The optional parameters starting from *`action`* form an activity (see `borg activity ...`) to be carried out when the user clicks on the notification. See the description of `borg alarm set` below for special treatment of the *`component`* parameter. The optional *`icon`* must be a PNG or JPG image encoded as base64 string. The size of the icon should be 24 by 24 pixels.

`borg notification delete` *`?id?`*

Deletes a notification identified that was created with the *`id`* specified. If no *`id`* is provided, all notifications are deleted.

`borg notification led` *`id argb onms offms`*

Adds a notification controlling the device LED. The integer *`id`*, specified by the caller, is used to identify the notification for later modification or deletion. The integer parameter *`argb`* is the LED color as combined RGB value with alpha channel, *`onms`* and *`offms`* are integers, too, controlling the duty cycle of blinking.

`borg vibrate` *`ms`*

Turns the vibration motor on for integer *`ms`* milliseconds.

`borg beep` *`?uri?`*

Plays a notification sound. If *`uri`* is specified and not an empty string, it is played as notification/ringtone/alarm sound. If given as empty string, the current playback is stopped. If omitted or unable to be resolved, the default notification sound is played. The URI typically has the pattern `content://media/{internal,external}/audio/media/<id>`, where `id` is an integer number identifying a sound file. The `borg content` command can be used to obtain information on notification sounds from Android's media provider.

`borg speak` *`text ?lang pitch rate?`*

Gets the Android to read out the string *`text`*. Optional parameter *`lang`* is the language code for the spoken language, e.g. en, en_US, de, es, etc. Optional parameters *`pitch`* and *`rate`* control the voice and speed as float values. On success an integer number >= 1000 is returned which identifies the text to be spoken in various virtual events. On error a negative number is returned. More information.

`borg stopspeak`

Stops speech output.

`borg isspeaking`

Returns a small integer indicating the state of speech output. Zero indicates initialization of speech output has been performed but no speech output is currently active. One is returned when speech output is

active. A small negative number indicates an error, temporary or permanent unavailability of the text-to-speech facility. In order to start up the text-to-speech facility this command can be used. The first call usually returns -1 and calls some few hundred milliseconds later return zero, indicating availability of the text-to-speech facility.

`borg endspeak`

Stops speech output and releases system resources.

`borg toast` *`text ?flag?`*

Displays a text notification *`text`* for a short period of time. The duration of that display is somewhat longer when *`flag`* is specified as true.

`borg spinner on|off`

Displays or withdraws a spinner (rotating symbol indicating busy state) depending on argument.

Location-Related Commands

`borg location start` *`?minrate-in-ms ?min-dist-in-m??`*

Begins acquiring location data via the Android OS (which may choose to use GPS, network info, etc.).

`borg location stop`

Ends location data acquisition.

`borg location get`

Returns the location data (as an `array set` or `dict create`form) where the key is the location source. Location updates trigger a virtual event `<<LocationUpdate>>` that is sent to all toplevel widgets. These toplevel event-handlers should, in turn, invoke `borg location get` to refresh their knowledge.

`borg location gps`

Returns GPS information in `array set` or `dict create` form with the keys `state` (on or off) and `first_fix` (time in seconds until first fix expected). Updates in GPS information are indicated by the `<<GPSUpdate>>` virtual event sent to all toplevel widgets.

`borg location satellites`

Returns GPS satellite information in `array set` or `dict create` form where the key is a numerical index and the values are per satellite information in `array set` or `dict create` form containing the fields `index`, `azimuth`, `elevation`, `prn`, `snr`, `almanac`, `ephemeris`, and `infix`. Updates in GPS information are indicated by the `<<GPSUpdate>>` virtual event sent to all toplevel widgets.

`borg location nmea`

Returns a string made up of the NMEA sentences collected over the last second. Updates in this string are indicated by the `<<NMEAUpdate>>` virtual event sent to all toplevel widgets.

System-Related Commands

`borg displaymetrics`

Returns information about the display in form suited for `array set` or `dict create`, e.g. display resolution, pixel density. The entry `rotation` gives the current screen rotation in degrees. The 0 degree point varies between devices, typical smart phones report 0 for portrait, tablets report 0 for landscape orientation. More information.

`borg osbuildinfo`

Returns information about the operating system and device in form suited for `array set` or `dict create`, e.g. Android API level, version, device name, manufacturer etc. More information.

`borg queryfeatures`

Returns information about features of the system (a lengthy list of strings) which is obtained from getSystemAvailableFeatures.

`borg queryconsts` *`classname`*

Returns a dictionary of constants of the (loaded) Java class *`classname`*. The keys are the names of the constants, the values their value. For example, the symbols of `SYSTEM_UI_*` flags are available when you evaluate:

```
borg queryconsts android.view.View
```

`borg queryfields` *`classname`*

Returns a dictionary of constants and static fields of the (loaded) Java class *classname*. This is similar to `borg queryconsts` but allows to retrieve no-constant strings, too. Most useful in combination with the android.os.Environment class.

`borg packageinfo ?name?`

Returns information about installed packages or an individual package if its *name* is given. In the first case, a list with package names is returned, in the latter case a list of key value pairs with package information which can be used for `array set` or `dict create`.

`borg providerinfo`

Returns the authority names of all content providers known to the system as a list.

`borg log prio tag message`

Writes the message *message* to Android's system log with priority *prio* (one of `verbose`, `debug`, `info`, `warn`, `error`, or `fatal`) and a user chosen prefix *tag*. These log messages can be displayed using `adb logcat` on the development system.

`borg trace message script`

Evaluates *script* and adds *message* before and after that evaluation to Android's system trace buffer. This is supported only on newer Android OS versions (4.3 and above) and further described in the android.os.Trace class.

`borg systemproperties ?name?`

Returns a list of system properties (a lengthy list of key value pairs) or the value of a specific system property if *name* is given.

`borg checkpermission name`

Returns true or false depending on manifest permission *name*. For a detailed list see the description of the android.Manifest.permission class.

Sensor-Related Commands

`borg sensor list`

Returns a list of the available sensors of the device. Each item is suited for `array set` or `dict create` and contains the fields `index` (integer index of the sensor, used to identify it), `type` (sensor type, one of `accelerometer`, `temperature`, `game_rotation_vector`, `geomagnetic_rotation_vector`, `gravity`, `gyroscope`, `gyroscope_uncalibrated`, `light`, `linear_acceleration`, `magnetic_field`, `magnetic_field_uncalibrated`, `orientation`, `pressure`, `proximity`, `relative_humidity`, `rotation_vector`, `step_counter`, and `step_detector`), `mindelay` (minimum update interval in milliseconds), `maxrange` (maximum range, floating point), `resolution` (floating point), `power` (in mA, floating point), and `name` (name of the sensor). More information.

`borg sensor enable|disable|state index`

Turns the sensor identified by *index* on or off, or returns its state (0=off, 1=on). An enabled sensor generates `<<SensorUpdate>>` virtual events which are sent to toplevel windows. These events are either periodic updates or change notifications depending on the kind of sensor and its refresh rate. If a sensor is not read out using `borg sensor get ...` for a certain amount of time that sensor is automatically disabled to conserve battery power. If the application enters background (see virtual event `<<WillEnterBackground>>`) all enabled sensors are disabled and re-enabled again when the application comes back to foreground (see virtual event `<<WillEnterForeground>>`).

`borg sensor get index`

Returns the last value acquired from the sensor identified by *index* as a list suited for `array set` or `dict create` containing the fields `index` (integer), `enabled` (sensor state, 0 or 1), `maxrange` (see above), `resolution` (see above), `accuracy` (the accuracy of this value), `values` (the sensor value, zero or more floating point numbers). When both `accelerometer` and `magnetic_field` sensors are turned on, the information for the `magnetic_field` sensor has two additional entries `orientation` (3 element list of azimuth, pitch, roll) and `inclination`. The `pressure` sensor has an additional entry `altitude` (meters above sea level). More information.

Android Content (shared databases)

`borg content query uri ?columns ?where ?whereargs ?orderby????`

Performs a query on an Android content provider given by *uri* and returns a cursor token (a Tcl command which deals with that cursor). The optional *columns* are a list of database columns to appear in the result set, e.g.

```
set cursor [borg content query content://contacts/people {display_name _id}]
```

where an empty list of database columns works like the SQL statement "SELECT *".

The optional *where* and *whereargs* parameters form the SQL WHERE clause of the query. Question mark parameter markers in *where* are positionally substituted by the information from the *whereargs* list. If the *where* parameter does not use a question mark parameter marker, an empty parameter string is obligatory as in

```
set cursor [borg content query content://contacts/people {} "_id=810" {}]
```

which is equivalent to

```
set cursor [borg content query content://contacts/people {} "_id=?" 810]
```

The optional *orderby* forms the SQL ORDERBY part of the query.

Here's another example demonstrating how data is retrieved:

```
set cursor [borg content query content://settings/system]
# initially, cursor points before first row
while {[$cursor move 1]} {
    puts [$cursor getrow]
}
```

`borg content delete` *uri selection ?value ...?*

Deletes rows from an Android content provider given by *uri* and returns the number of deleted rows. *selection* forms the SQL WHERE clause of the deletion where question mark parameter markers are substituted by *value*. Example:

```
borg content delete content://settings/system name=? my_item
```

`borg content insert` *uri key value ...*

Inserts a row into an Android content provider given by *uri* and returns another URI which identifies the inserted row. *key* and *value* are pairs of column name and column value for the SQL INSERT operation. Example:

```
borg content insert content://settings/system name my_item value "Some value"
-> content://settings/system/4711
```

`borg content update` *uri values ?selection ?args??*

Updates zero or more rows of an Android content provider given by *uri* and returns the number of updated rows. *values* is a list made up of a sequence of column names and column values. *selection* is the optional SQL WHERE clause with question mark parameter markers. The parameter markers are substituted by the values from *args*. Example:

```
# system settings wants the name in the URI
borg content update content://settings/system/my_item {value {New Value}}
```

Cursors from Android Content queries

When a `borg content` query returned a cursor token, this token is a Tcl command to deal with the query's result set:

`$cursor close`

Finishes the query and deletes the Tcl command.

`$cursor columnnames`

Returns a list of column names of the result set.

`$cursor count`

Returns the number of rows of the result set.

`$cursor getblob` *index*

Returns the *index*th column (zero-based) of the current row of the result set as a base64 encoded string.

`$cursor getdouble` *index*

Returns the *index*th column (zero-based) of the current row of the result set as a floating point number.

`$cursor getint` *index*

Returns the *index*th column (zero-based) of the current row of the result set as a integer number.

`$cursor getpos`

Returns the index of the current row (zero-based).

`$cursor getrow`

Returns the current row as a Tcl list made up of column names and values, like `{name1 value1 name2 value2}` in the order specified by the cursor's constructor. Blobs from the result set are converted into base64 encoded strings.

`$cursor getstring` *`index`*

Returns the *`index`*th column (zero-based) of the current row of the result set as a string.

`$cursor isnull` *`index`*

Returns true when the *`index`*th column (zero-based) of the current row of the result set is an SQL NULL value.

`$cursor move` *`pos`*

Relative move of the current row index. Negative *`pos`* goes backward.

`$cursor moveto` *`pos`*

Absolute move of the current row index.

Speech Recognition

`borg speechrecognition intent` *`args cmd`*

Use the speech recognition service. *`args`* is a parameter list to control the speech recognition (more information). *`cmd`* is invoked when the speech recognition is complete. That procedure receives the parameters `retcode` and `data` as in the callback for `borg activity`. `data` is a list of key value pairs suited for `array set` or `dict create`.

`borg speechrecognition callback` *`cmd`*

Establishes *`cmd`* as global callback procedure receiving speech recognition events as described by the RecognitionListener interface (more information). That procedure receives the parameters `retcode` and `data` as in the callback for `borg activity`. `data` is a list of key value pairs suited for `array set` or `dict create`. There are up to two special keys present in data: `type` giving the event type (one of `result`, `partialresult`, `ready`, `event`. `rms`, `end`, `begin`) and `value` giving numeric values for certain event types (error code for `error`, audio level for `rms`).

`borg speechrecognition start` *`args`*

Starts the speech recognition. For *`args`* see the description in `borg speechrecognition intent .....`

`borg speechrecognition cancel`

Immediately cancels the speech recognition. No further events are reported through the global callback procedure.

`borg speechrecognition stop`

Stops the speech recognition. Events can be still reported through the global callback procedure.

Telephone-Related Commands

`borg phoneinfo`

Returns information about the current state of the telephone as a list suited for `array set` or `dict create`. This information is only available when the application manifest has the `android.permission.READ_PHONE_STATE` permission (which is left out in current AndroWish releases). For further information see the Android documentation on the classes TelephonyManager, PhoneStateListener, and SignalStrength.

`borg sendsms phonenumber msg ?`*`action_send action_delivered smsc`*`?`

Sends an SMS text message *`msg`* to *`phonenumber`*. The optional arguments *`action_send`* and *`action_delivered`* are the action names of broadcast intents which are generated on state changes regarding the SMS message and can be captured by a broadcast listener callback (see `borg broadcast register` et.al.). The optional argument *`smsc`* is the SMS message center. The command returns 1 on successful start of the send operation, 0 otherwise. It is only available when the application manifest has the `android.permission.SEND_SMS` permission (which is left out in current AndroWish releases). For further information see the Android documentation on the class SmsManager.

Broadcast

`borg broadcast list ?`*`action`*`?`

Returns a list of all registered broadcast handlers in the interpreter when *action* is omitted. Otherwise it returns the command to be invoked when the broadcast *action* is received.

`borg broadcast register` *action cmd*

Registers the command *cmd* to be invoked when the broadcast *action* is received.

`borg broadcast unregister` *action*

Unregisters the command bound to the reception of the broadcast *action*.

`borg broadcast send` *action* ?*uri type categories arguments*?

Sends the broadcast *action* with the optional properties/arguments *uri*, *type*, *categories*, and *arguments*. For the optional items see the description in `borg activity`.

Locale

`borg locale ?default?`

Returns information about the current default locale of the JVM. The result is in a form suitable for `array set` or `dict get` and contains the fields `country`, `display_country`, `display_language`, `display_name`, `display_variant`, `iso3_country`, `iso3_language`, `language`, and `variant`.

`borg locale` *lang*

Returns information about the locale identified by *lang*, which must be specified as a two letter code with an optional variant and an optional encoding part, e.g. `de`, `fr_BE`, or `en_GB.UTF-8`.

`borg locale tts`

Returns information about the locale used for text-to-speech. If text-to-speech facilities weren't used when the command is invoked, the returned information is identical with `borg locale default`.

`borg locale set` *lang*

Changes the current locale of the JVM to the language code *lang*. If the locale change succeeded, the environment variable `env(LANG)` is changed accordingly.

Camera-Related Commands

Camera support is available on devices with Android 3.0 or newer.

`borg camera close`

Close the camera. Returns non-zero on success, zero otherwise, e.g. when the camera was already closed.

`borg camera current`

Returns the currently opened camera number or -1 when the camera is not opened. On many tablets camera 0 is the back-facing camera, and camera 1 the front-facing one.

`borg camera grayimage` ?*photo*?
`borg camera greyimage` ?*photo*?

Copies the most recent camera preview as grey image into the photo image identified by *photo*. Returns non-zero on success or zero if no data transfer has taken place. If *photo* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel, and the last the image's grey values with 1 byte per pixel as a byte array. In this case an error is indicated by throwing an exception. An experimental feature is direct rendering into a widget. In this case the *photo* parameter must be the path name of a Tk window which should be a frame or toplevel widget. When the camera is started the background color of the widget should be set to an empty string so that no drawing calls from Tk are carried out. When the camera is stopped, it should be set to black.

`borg camera image` ?*photo*?

Copies the most recent camera preview as color image into the photo image identified by *photo*. Returns non-zero on success or zero if no data transfer has taken place. If *photo* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel, and the last the image's RGB values with 3 bytes per pixel in red, green, blue order as a byte array. In this case an error is indicated by throwing an exception. An experimental feature is direct rendering into a widget. In this case the *photo* parameter must be the path name of a Tk window which should be a frame or toplevel widget. When the camera is started the background color of the widget should be set to an empty string so that no drawing calls from Tk are carried out. When the camera is stopped, it should be set to black.

`borg camera info`

Returns information about the currently opened camera as a two element list made up of integer numbers. The first is the rotation of the camera relative to the screen, the second an indication for front-facing (1) or

back-facing (0) view of the camera relative to the screen. If no camera is opened the result is an empty list.

`borg camera jpeg`

Returns a JPEG image of the camera as a byte array after preview has been started using `borg camera start` and JPEG capture has been initiated with `borg camera takejpeg`. In contrast to `borg camera image ...` this command consumes the image. If no JPEG picture is available when the command is invoked, an error is thrown.

`borg camera mirror ?`*`x y`*`?`

Controls mirroring of preview images which are mirrored along the X axis when *x* is one and along the Y axis when *y* is one. `borg camera mirror 0 1` is useful to mirror the preview image of a front-facing camera.

`borg camera numcameras`

Returns the number of available cameras.

`borg camera open ?`*`num`*`?`

Opens camera number *num* and returns non-zero on success. Only one camera can be opened at any one time. On error or when a camera is already opened, zero is returned. When *num* is omitted the first camera is opened (usually the back-facing if two cameras are available).

`borg camera orientation ?`*`degrees`*`?`

Returns the current orientation of the preview image relative to the screen or changes it to *degrees*.

`borg camera parameters ?`*`key value ...`*`?`

Returns or changes camera parameters given as key-value pairs, e.g. `preview-size 320x240` will change the size of preview images to width 320 and height 240. The command returns the current camera parameters (after the potential change, when keys and values where given) as a key-value list which can be processed with `array set` or `dict get`.

`borg camera start`

Starts the camera. Acquired preview images are reported by the virtual event `<<ImageCapture>>`. Returns non-zero on success, zero when the camera is already started or an error has been detected. When the acquisition of camera preview images is running `borg camera image` or `borg camera greyimage` must be invoked within 5 seconds, otherwise image acquisition is automatically stopped and needs to be restarted with another `borg camera start` command.

`borg camera state`

Returns the current camera state as string: `unknown`, `closed`, `stopped`, or `capture`.

`borg camera stop`

Stops the camera, i.e. no more images are acquired. Returns non-zero on success, zero when the camera is already stopped or an error has been detected.

`borg camera takejpeg`

Requests the camera to take a JPEG image. It is required that the camera is capturing, i.e. `borg camera start` has been called already. The point in time when acquisition of the JPEG image starts is indicated by the virtual event `<<Shutter>>`. When the JPEG image is ready for processing the virtual event `<<PictureTaken>>` is sent. The command returns a non-zero value when JPEG capture is in progress, zero on error.

NFC Related

Many devices have hardware support for NFC (Near Field Communication) tags. In order to deal with such items, a callback command for the broadcast `tk.tcl.wish.nfc` must be registered. The callback's arguments contain information on the NFC tag in these keys:

`android.nfc.extra.ID`

The the base64 encoded ID of the tag.

`android.nfc.extra.TAG`

The underlying/supported technologies of the tag as a string. Currently only `android.nfc.tech.Ndef` and `android.nfc.tech.NdefFormatable` are detected and handled.

`android.nfc.extra.NDEF_MESSAGES`

If present contains the NDEF formatted information contained in the tag encoded in base64.

The last read tag ID is remembered and can be dealt with using these `borg` subcommands:

`borg ndefread` *`tagid ?cached?`*

Returns the current or *`cached`* NDEF formatted information contained in the tag given *`tagid`* as base64 encoded string.

`borg ndefwrite` *`tagid ndefmsg`*

Writes the NDEF formatted information (one or more NDEF records) in *`ndefmsg`* which must be base64 encoded into the tag given *`tagid`*.

`borg ndefformat` *`tagid ndefmsg`*

Formats an empty tag and like `borg ndefwrite` writes NDEF formatted information into the tag. An unformatted tag can be detected in the `tk.tcl.wish.nfc` callback procedure by inspecting the technology information: The string `android.nfc.tech.Ndef` is absent but the string `android.nfc.tech.NdefFormatable` is present.

OS Environment

Information provided by the android.os.Environment class.

`borg osenvironment datadir`

Return the user data directory (see getDataDirectory).

`borg osenvironment downloadcachedir`

Return the download/cache content directory (see getDownloadCacheDirectory).

`borg osenvironment externalstoragedir`

Return the primary shared/external storage directory (see getExternalStorageDirectory).

`borg osenvironment externalstoragepublicdir ?`*`type`*`?`

Get a top-level shared/external storage directory for placing files of a particular type (see getExternalStoragePublicDirectory). The parameter *`type`* can be obtained by using information returned from `borg queryfields android.os.Environment`.

`borg osenvironment externalstoragestate`

Returns the current state of the primary shared/external storage media (see getExternalStorageState).

`borg osenvironment isexternalstorageemulated`

Returns whether the primary shared/external storage media is emulated (see isExternalStorageEmulated).

`borg osenvironment isexternalstorageremovable`

Returns whether the primary shared/external storage media is physically removable (see isExternalStorageRemovable).

`borg osenvironment rootdir`

Return root of the "system" partition holding the core Android OS (see getRootDirectory).

Shared Preferences

An interface to android.content.SharedPreferences is provided using the `borg sharedpreferences` subcommand. This allows to load/store typed values of an application in a key-value store which does not require extra file permissions.

`borg sharedpreferences` *`file`* `getboolean` *`key default`*

Return a boolean value (using *`default`* if not present) from the shared preference file identified by *`file`* stored under the name *`key`*.

`borg sharedpreferences` *`file`* `getfloat` *`key default`*

Return a floating point value (using *`default`* if not present) from the shared preference file identified by *`file`* stored under the name *`key`*.

`borg sharedpreferences` *`file`* `getint` *`key default`*

Return an integer value (using *`default`* if not present) from the shared preference file identified by *`file`* stored under the name *`key`*.

`borg sharedpreferences` *`file`* `getlong` *`key default`*

Return a 64 bit integer value (using *`default`* if not present) from the shared preference file identified by *`file`* stored under the name *`key`*.

`borg sharedpreferences` *`file`* `getstring` *`key default`*

Return a string value (using *`default`* if not present) from the shared preference file identified by *`file`* stored under the name *`key`*.

`borg sharedpreferences` *`file`* `setboolean` *`key value`*

Store the boolean *`value`* into the shared preference file identified by *`file`* under the name *`key`*.

`borg sharedpreferences` *`file`* `setfloat` *`key value`*

Store the floating point *`value`* into the shared preference file identified by *`file`* under the name *`key`*.

`borg sharedpreferences` *`file`* `setint` *`key value`*

Store the integer *`value`* into the shared preference file identified by *`file`* under the name *`key`*.

`borg sharedpreferences` *`file`* `setlong` *`key value`*

Store the 64 bit integer *`value`* into the shared preference file identified by *`file`* under the name *`key`*.

`borg sharedpreferences` *`file`* `setstring` *`key value`*

Store the string *`value`* into the shared preference file identified by *`file`* under the name *`key`*.

`borg sharedpreferences` *`file`* `remove` *`key`*

Remove the value stored under the name *`key`* from the shared preference file identified by *`file`*.

`borg sharedpreferences` *`file`* `clear`

Remove all key-value pairs stored in the shared preference file identified by *`file`*.

`borg sharedpreferences` *`file`* `all`

Return a Tcl list suitable for `array set` of all key-value pairs from the shared preference file identified by *`file`*.

`borg sharedpreferences` *`file`* `alltypes`

Return a Tcl list suitable for `array set` of all keys and their respective value types from the shared preference file identified by *`file`*.

`borg sharedpreferences` *`file`* `keys`

Return a Tcl list of all keys from the shared preference file identified by *`file`*.

General

`borg withdraw`

Hides the application window by putting it to the end of Android's activity stack. Can be useful when bound to the Back key (`<Key-Break>` in AndroWish). There's no opposite command, i.e. the application can be brought to front again only by user interaction.

`borg brightness` ?*`percent`*?

Sets or gets the screen brightness. If the percentage is negative, the default value is restored.

`borg onintent` ?*`command`*?

Sets or gets the callback *`command`* which is evaluated when the application received an Android intent. When evaluating that command, the parameters action name, URI, MIME type, categories, arguments from the intent are appended. When the callback is set for the first time, it gets immediately evaluated with the parameters of the current (startup) intent.

`borg queryactivites` *`action`* ?*`uri type categories component`*?

Queries Android's activity manager for activities on the given intent parameters. *`categories`* and *`component`* are optional lists. The latter when non-empty must contain the two elements package name and class name.

`borg queryservices` *`action`* ?*`uri type categories component`*?

Queries Android's activity manager for services on the given intent parameters, similar to `borg queryactivities`.

`borg querybroadcastreceivers` *`action`* ?*`uri type categories component`*?

Queries Android's activity manager for broadcast receivers on the given intent parameters, similar to `borg queryactivities`.

`borg screenorientation ?orient?`

Queries or switches the screen orientation, *orient* can be one of `unspecified`, `landscape`, `portrait`, `user`, `behind`, `sensor`, `nosensor`, `sensorlandscape`, `sensorportrait`, `reverselandscape`, `reverseportrait`, `fullsensor`, `userlandscape`, `userportrait`, `fulluser`, or `locked`.

`borg keyboardinfo`

Returns information about the current keyboard configuration as a list suited for `array set` or `dict create` with these fields: `keyboard` with possible values `none`, `12key`, and `qwerty`, `hidden` and `hard_hidden` with values 0 (not hidden), 1, (hidden), and -1 (unknown). This can be read out any time. An update in keyboard configuration state is indicated by the virtual event `<<KeyboardInfo>>`.

`borg alarm clear action ?uri type categories component?`

Clears an alarm whose pattern matches the given intent parameters.

`borg alarm set when repeat action ?uri type categories component arg ...?`

Sets an alarm to fire at *when* (UN*X epoch, seconds) with repetition each *repeat* seconds, when *repeat* is greater than 0. The alarm sends an intent made up of the given intent parameters (*action* etc.). The *component* parameter is interpreted specially: when empty no component is set on the intent, when given as self the calling package/class is set as component for the intent (sending the intent to itself, i.e. the callback of `borg onintent` will receive it). In all other cases component must be a list with the two elements package name and class name. *arg* and following parameters are added to the intent as key value pairs of extra data.

`borg alarm wakeup when repeat action ?uri type categories component arg ...?`

Like `borg alarm set` but this type of alarm is able to wake up the device when suspended (device behavior depends on lock screen settings).

`borg activity action uri type ?categories ?component ?arguments ?callback????`

This a very flexible command that allows extensive access to the Android OS and other applications. *categories*, *component*, and *arguments* are optional lists. *callback* is the name of the procedure that is evaluated when the activity *action* is complete. *arguments* are key-value pairs where the values are mapped to Java strings by default. If the key is a 2-element list made up of a data type indicator (`int`, `byte`, `short`, `char`, `long`, `float`, `double`, `Uri`) followed by the key, the value is converted to that data type. See below for some examples of this command.

`borg systemui ?flags?`

Returns or sets various flags to control certain aspects of system UI elements displayed on the device's screen. This is currently supported only on Android 4.4 and newer versions. See the documentation on android.view.View for a description of the `SYSTEM_UI_*` flags.

Events

These events are generated for/by certain `borg` related commands. They are reported to toplevel widgets only.

`<<LocationUpdate>>`

Location information has been updated and can be read out using `borg location get`.

`<<GPSUpdate>>`

GPS related information has been updated and can be read out using `borg location gps` and `borg location satellites`.

`<<NMEAUpdate>>`

NMEA data has been updated and can be read out using `borg location nmea`.

`<<NetworkInfo>>`

Network state has changed and can be read out using `borg networkinfo`.

`<<TetherInfo>>`

Tethering state has changed and can be read out using `borg tetherinfo`.

`<<Bluetooth>>`

A change in Bluetooth state and/or scan mode has occured and can be read out using `borg bluetooth state` and/or `borg bluetooth scanmode`.

`<<SensorUpdate>>`

A sensor can be read using `borg sensor get ...`. The field `%x` identifies the sensor.

`<<KeyboardInfo>>`

The keyboard configuration did change and can be obtained by `borg keyboardinfo`.

`<<PhoneCallState>>`

The call state of the telephone has changed and can be obtained by `borg phoneinfo`.

`<<PhoneDataActivity>>`

The state of the telephone's data state has changed and can be obtained by `borg phoneinfo`.

`<<PhoneConnectionState>>`

The state of the telephone's connectivity has changed and can be obtained by `borg phoneinfo`.

`<<PhoneServiceState>>`

The service state of the telephone has changed and can be obtained by `borg phoneinfo`.

`<<PhoneSignalStrength>>`

The signal quality of the telephone has changed and can be obtained by `borg phoneinfo`.

`<<ImageCapture>>`

A preview camera image is ready and can be obtained by `borg camera image` *photo* or `borg camera greyimage` *photo*. The field `%x` represents the camera capture state (true when preview images are captured, false when capture will be stopped).

`<<Shutter>>`

The camera is about to take a JPEG image.

`<<PictureTaken>>`

The camera has taken a JPEG picture which can be obtained and consumed by `borg camera jpeg`. When the event is reported, image capture of preview images is automatically stopped.

`<<USBAttached>>`

A USB device was attached. To find out information about the device, use the `borg usbdevices` command. This event is generated on Android 4.4 and newer.

`<<USBDetached>>`

An USB device was detached (opposite of `<<USBAttached>>`).

`<<TTSInit>>`

The text-to-speech facility has been started up or shut down. The %x substitution gives an indication for startup (=0), error (=-1), and unavailability (<-1). Supported in Android 4.1 and higher.

`<<TTSStart>>`

Speech output of a string has started. The %x substitution is equal to the integer returned by the corresponding `borg speak` command. Supported in Android 4.1 and higher.

`<<TTSError>>`

Error indication for a string to be spoken by `borg speak`. The %x substitution is equal to the integer returned by the corresponding `borg speak` command as for the `<<TTSStart>>` event. Supported in Android 4.1 and higher.

`<<TTSDone>>`

End of speech indication for a string to be spoken. The %x substitution is equal to the integer returned by the corresponding `borg speak` command as for the `<<TTSStart>>` event. Supported in Android 4.1 and higher.

`borg activity` Examples

Sample code to open a browser on the Tcl'ers wiki:

```
    borg activity android.intent.action.VIEW http://wiki.tcl.tk text/html
```

Sample code to launch the "wifi settings" page:

```
borg activity android.settings.WIFI_SETTINGS {} {} {} {} {}
```

Sample code to update the Androwish application from its APK:

```
borg activity android.intent.action.VIEW "file:///sdcard/androwish.apk"
application/vnd.android.package-archive
```

Sample code to capture an image (only makes thumbnails):

```
    proc callback {retcode action uri mimetype categories data} {
        if {$retcode == -1} {
            # SUCCESS
            array set result $data
            if {[info exists result(data)]} {
                myphoto configure -data $result(data)
            }
        }
    }
    package require Img
    image create photo myphoto
    borg activity android.media.action.IMAGE_CAPTURE {} {} {} {} {} callback
```

Sample code to capture an image, which makes full size images but requires a file on external storage:

```
    proc callback {filename retcode action uri mimetype categories data} {
        if {$retcode == -1} {
            # SUCCESS
            myphoto configure -file $filename
            catch {file delete -force $filename}
        }
    }
    package require Img
    image create photo myphoto
    set filename [file join $env(EXTERNAL_FILES) myphoto.jpeg]
    borg activity android.media.action.IMAGE_CAPTURE {} {} {} {} \
       [list {Uri output} file://$filename] [list callback $filename]
```

Reading barcodes using the ZXing barcode scanner (which needs to be be installed on your device):

```
    proc barcode_read {code action uri type cat data} {
        array set result $data
        if {[info exists result(SCAN_RESULT)]} {
            # that is the barcode
            # result(SCAN_RESULT_FORMAT) is the barcode format
        }
    }

    borg activity com.google.zxing.client.android.SCAN {} {} {} {} {} barcode_read
```

AndroWish SDK

The AndroWish Software Development Kit

This is a preliminary description of the AndroWish SDK. It consists of a large ZIP file made up of prebuilt components (Java classes, shared libraries, Tcl library files, and other resource and property files) and a small graphical tool called **bones** to customize these components and to finally create an installable Android package (an APK file).

Thus, in theory it is not necessary anymore to mess with the many pieces of source code which make up AndroWish but to boldly click with the mouse some ten times to get a Tcl based Android App.

Prerequisites

- A recent Java Development Kit, version 1.6 or 1.7
- Android Standalone SDK tools.
- Optionally Apache ant, often available as an optional installable package of a Linux distribution
- Tcl/Tk **wish** version 8.5 or 8.6 (preferred) e.g. from ActiveState's ActiveTcl, but like **ant**, often available as an optional installable package of a Linux distribution

AndroWish SDK Setup

The current AndroWish SDK has been tested on some Linux distributions (CentOS 6, Linux Mint 17.1) and on Windows 7 (32 bit). It is reported to be usable on MacOSX, too.

Download the current AWSDK.zip and unpack it. Ensure, that you've set up your environment and/or path so that the Android SDK tools can be found by the build tools (**gradle** or **ant**). Ensure, that you can run Tcl programs using **wish**.

Experimental! For the adventurous there's a single file Win32 (32 bit) binary in bones.exe which contains both a current **wish** and AWSDK.zip. It should be copied to a directory of its own where it unpacks the built-in AWSDK.zip when executed for the first time.

Directory Structure of the SDK

After the AWZIP.zip has been unpacked, the resulting AWSDK directory contains these files and directories (only the most important ones shown):

File/Directory	Remarks
AndroidManifest.xml	App descriptor, read/written by the **bones** tool. Contains App's entry point and permissions.
ant.properties	Build information for **gradle** or **ant**. Read/written by the **bones** tool for keystore information (code signing).
assets/*	Tcl/Tk libraries and additional support files (e.g. version information and package inventory). Content controlled by the **bones** tool.
assets/app/*	User code. The file **main.tcl** is automatically run by the App. Other user/App specific files should go here, too.
build.gradle	Control file for **gradle** (like Makefile for **make**).
build.xml	Control file for **ant** (like Makefile for **make**).
_casket/*	Directory where the **bones** tool moves and keeps track of unselected optional components (Tcl/Tk and native shared libraries).
gradle/*	Wrapper/support files for **gradle**.
gradlew	Shell script to run **gradle** on UN*X platforms.
gradlew.bat	Batch file to run **gradle** on Windows platforms.
libs/*.jar	Precompiled Java libraries built into the App.
libs/armeabi/*.so	Precompiled native shared libraries for ARM processors. Content controlled by the **bones** tool.
libs/x86/*.so	Precompiled native shared libraries for x86 processors. Content controlled by the **bones** tool.
local.properties	Information for **ant** to locate the Android SDK. Updated once on first run of the **bones** tool.
res/*	App resources, e.g. PNG icon files in various resolutions. Modified by the **bones** tool for App icons.
settings.gradle	Project settings for **gradle**.

src/*	Java sources, App entry point (an empty Java class deriving from the AndroWish activity super class). Modified by the **bones** tool according to the user chosen package/class names.
tools/bones	Tcl source of the **bones** tool

Except for the "assets/app" directory the layout and content of the AndroWish SDK directory tree should not be altered manually in order to not confuse the **bones** tool.

External Tools

The following table lists the external programs which are used throughout operation of the **bones** tool.

Program	Location	Remarks
adb	$ANDROID_HOME/platform-tools/adb (Unix) %ANDROID_HOME%/platform-tools/adb (Win32) adb (fallback, all)	Android Debug Bridge used to optionally install final package and to start it on device or emulator.
android	$ANDROID_HOME/tools/android (Unix) %ANDROID_HOME%/tools/android (Win32) android (fallback, all)	Android SDK Platform Manager used to setup project initially. Required.
ant	$ANT_HOME/bin/ant (Unix) %ANT_HOME%/bin/ant (Win32) ant (fallback, all)	Apache **ant** used to control the APK build process. Optional, deprecated.
fossil	fossil	Fossil repository program, optionally used on startup to verify state of source tree.
keytool	keytool	Key tool from the Java Development Kit. Only used when an new keystore for code signing is to be generated.

So the **bones** tool prefers to find the essential external programs using the two environment variables **ANDROID_HOME** and **ANT_HOME**, and the common fallback strategy is to search for the external programs using the normal search path for executable programs.

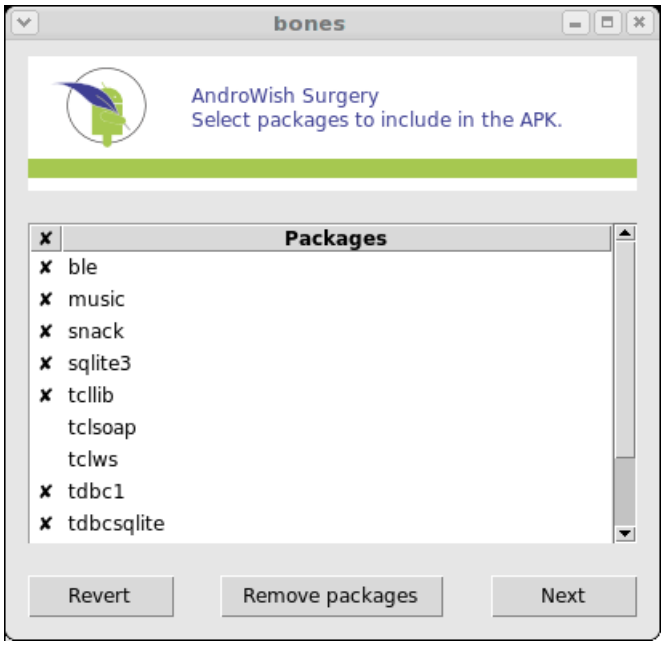
Since AndroWish version "Asteroid Day (2018-06-30)" **ant** is deprecated and **gradle** is used instead. To switch back to using **ant** the two files **gradlew** and **gradlew.bat** can be renamed in order to force the **bones** tool to fall back to **ant**.

Start the bones Tool

```
$ wish <path-where-AWSDK.zip-has-been-unpacked-to>/tools/bones
```

Fraction 1: Package Selection

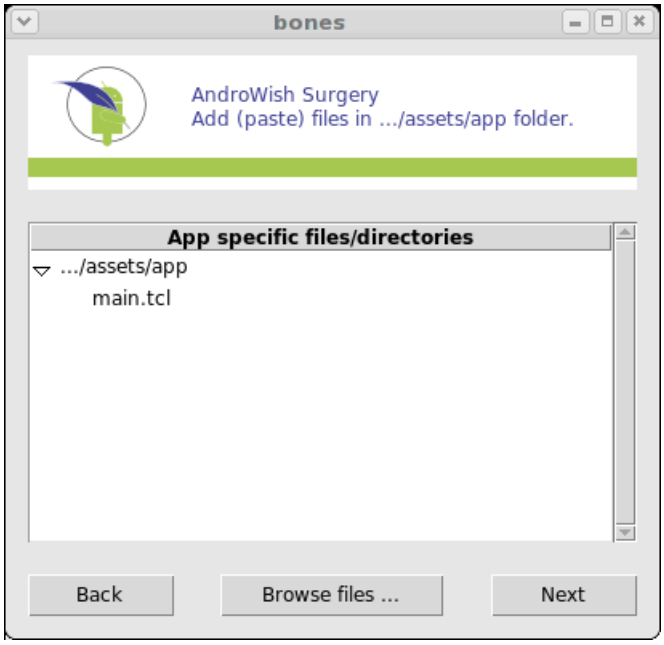
The first page of **bones** allows to remove optional pieces of AndroWish. An overview of included components gives Batteries Included. Uncheck unneeded packages in the list and remove them by pressing the **Remove packages** button. This moves the selected packages out of the staging area that they will not be added later to the APK. The **Revert** button moves all removed packages back to the staging area. When satisfied with your choice press the **Next** button for the next page.



When all optional components are omitted and CPU support is limited to ARM only (described below), the size of the resulting APK can be shrinked down to about 4 MByte instead of nearly 30 MByte with everything included.

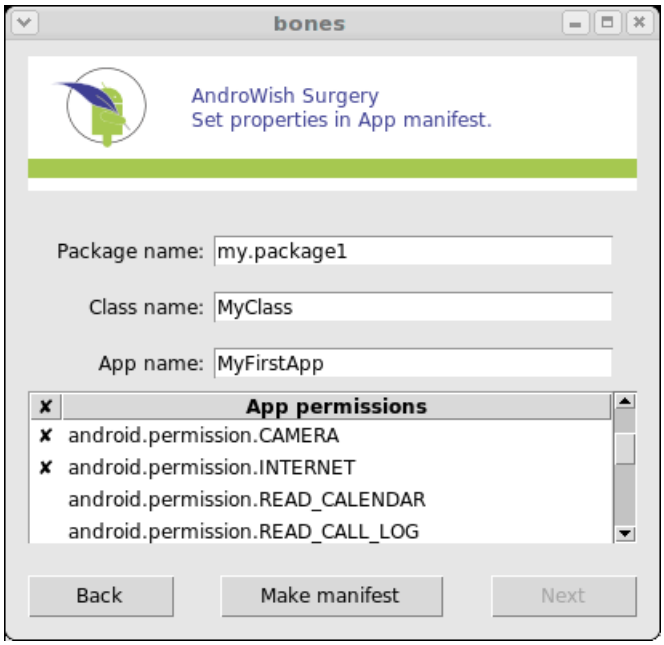
Fraction 2: Add App Specific Files

This page allows to add files to the folder **assets/app** within the staging area. The most important script is **main.tcl** which when found gets sourced on start up of AndroWish and thus allows to make your own App. Use the right mouse button to paste or deleted files and directories into the **assets/app** folder. This works from file managers which place their selected files into the clipboard. Otherwise use the **Browse files ...** button to open a simple file browser. When finished with this step press the **Next** button to continue.



Fraction 3: The App Manifest

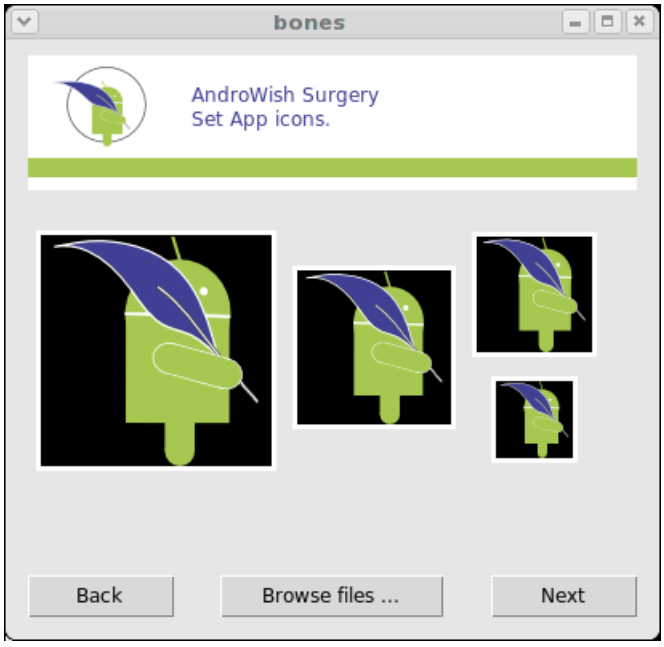
The manifest file (**AndroidManifest.xml**) is the application descriptor of the APK. It describes the App's unique name (the combination of **Package name** and **Class name**) and its label on the home screen (the **App name**). A good choice is a reversed domain name plus an optional package name component plus the final class name (example: the XZing barcode scanner is **com.google.zxing.BarcodeScanner**, i.e. **com.google.zxing** is the package, and **BarcodeScanner** the class name). The list box with **App permissions** allows to grant or revoke specific access permissions on device components. When you're satisfied with your selection press **Make manifest** to write your settings into the **AndroidManifest.xml**. After the new manifest settings have been written the **Next** button becomes sensitive to switch to the next page.



NB: Under the hood when writing the **AndroidManifest.xml** an additional directory tree with a Java file based on the package/class name fields is written, too, which contains an empty class definition deriving from **tk.tcl.wish.AndroWish**. This is the App's worm hole from the Java universe into our little Tcl/Tk galaxy. Its singular purpose is to satisfy the App naming requirement imposed by the Android empire.

Fraction 3a: Set App Icons

The icon(s) of the App shown in the home screen or in the notification area can be changed by pasting PNG files into the image labels. New icons should be provided in the four sizes 144x144 (XXHDPI), 96x96 (XHDPI), 72x72 (HDPI), and 48x48 (MDPI). When finished with the icons press the **Next** button to switch to the next page.

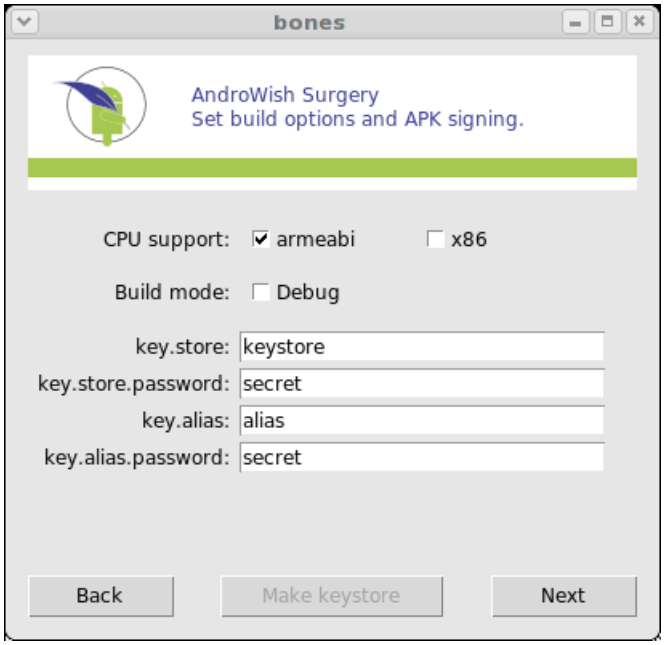


Fraction 4: Build Options, Code Signing

The **CPU support** selects which shared libraries are packaged into the APK. Currently, prebuilt support for ARM and x86 exists. However, latest experiments with various x86 devices (Intel Atom) showed, that usually ARM CPU support is sufficient since the x86 devices have an ARM emulation built in. Omitting x86 support squeezes about 7 MByte out of the APK when all possible packages have been selected.

The **Build mode** and various key store related fields control if a debug APK shall be built (which is signed with a special debug key). Alternatively, your own key shall be used to sign the APK. To create a key store from scratch enter the values as shown in the image and press the **Make keystore** button. Otherwise use an already existing keystore (e.g. **~/.keystore** , the default of Java's **keytool**) with approriate values for the key alias and passwords.

When ready for the final APK build step, press the **Next** button.



Fraction 5: APK Building

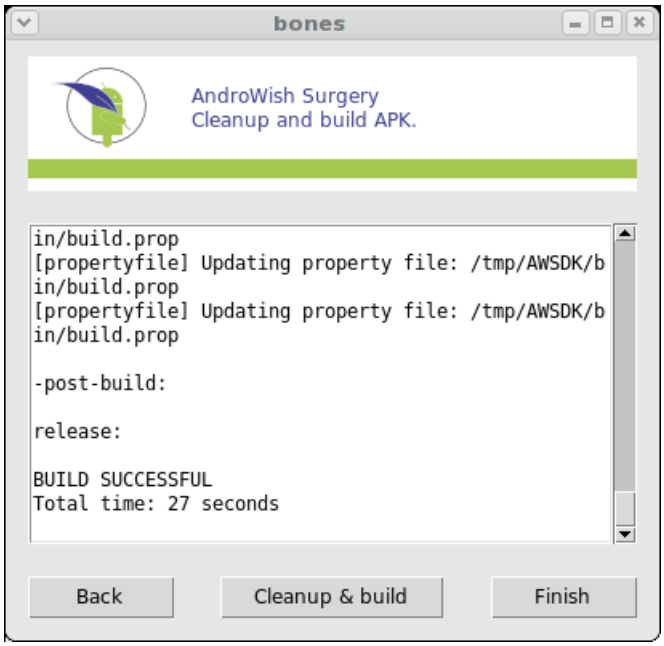
Building the APK (the **Cleanup & build** button) is equivalent to invoking

```
$ gradlew clean assembleDebug|assembleRelease
```

or

```
$ ant clean debug|release
```

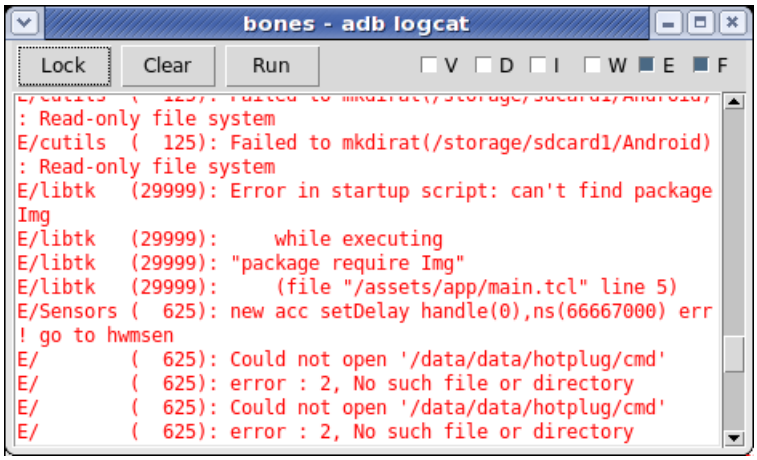
on the command line within the install directory of the AndroWish SDK. If everything went well, one of the last lines of output should read **BUILD SUCCESSFUL**. In this case the APK file can be found in **.../build/outputs/apk/AndroWishApp-debug.apk** for a **gradle** debug build, **.../bin/AndroWishApp-debug.apk** for an **ant** debug build, **.../build/outputs/apk/AndroWishApp-release.apk** for a **gradle** release build, or **.../bin/AndroWishApp-release.apk** for an **ant** release build, and be transferred to a device or emulator. If an Android device is connected to your development system the middle button changes to **Install & run** and allows to install and start the new APK when clicked.



Fraction 5a: Installing/Running the APK

The **Install & run** button opens a log window which displays the output of **adb logcat** (the log facility of the Android

Debug Bridge). An example is shown below which displays an error message originating from the App (the lines with "libtk" showing a Tcl error message). The check buttons with the single capital letters can be used to filter the log output according to its log level, e.g. "V" for verbose, "D" for debug, "E" for error. The **Lock/Scroll** button disables scrolling of the output window, the **Clear** button clears the output window, and the **Run** button allows to restart the App on the device or emulator.



Happy Tcl'ing

Batteries Included

Batteries Included

Following table lists the extensions built into AndroWish and/or undroidwish including pointers to project pages and/or documentation. Most extension names in the left most column can be used as package name in `package require`. The extension name is linked to the respective folder in the source tree. Column **A** shows availability in AndroWish, columns **W**/**O**/**L** in undroidwish (**W**indows, Mac**O**SX, and/or **L**inux). A minus sign indicates an extension which can't be provided for the respective platform for technical reasons. Column **B** indicates a binary package which needs to be compiled for the respective platform.

Many extensions also run on POT (plain old Tk, i.e. X11 based on POSIX, Win32 based on Windows, Cocoa based on MacOSX). However, there are some exceptions: BLT and Tkzinc are not ready for MacOSX. tcluvc is currently POSIX only and depends on an USB stack providing isochronous transfers. v4l2 requires a Video 4 Linux 2 infrastructure, which is available only for Linux and *BSDs.

Extension Name	B	Version	A	W	O	L	Remarks, URL, etc.
autoopts		0.6.1	✔	✔	✔	✔	Tcl module that automatically gives your program a command line interface, see https://gitlab.com/dbohdan/autoopts
awthemes		10.4	✔	✔	✔	✔	Brad Lanam's awlight, awdark, and black themes using tksvg, see https://sourceforge.net/projects/tcl-awthemes
ble	•	1.0	✔	-	-	-	Bluetooth Low Energy support, part of AndroWish, see ble command
BLT	•	2.4z	✔	✔	✔	✔	2D graph, bargraph, stripchart widgets, i.e. a subset of full BLT, http://sourceforge.net/projects/blt
bonjour	•	1.1			✔	✔	Tcl interface to Apple's implementation of the ZeroConf protocol, https://github.com/dongola7/tcl_bonjour/
borg	•	1.0	✔	-	-	-	Android integration, part of AndroWish, see Android facilities
BSD	•	1.9.2		-	✔	✔	FlightAware's package to various BSD UNIX system calls and library routines, https://github.com/flightaware/tclbsd
BWidget		1.9.16	✔	✔	✔	✔	Mega widget package, http://core.tcl-lang.org/bwidget
can2svg		0.3	✔	✔	✔	✔	Tk canvas to SVG conversion from https://thecoccinella.org
calc		?	✔	✔	✔	✔	Andy Goth's Tcl/Tk calculator application from the Tcl'ers Wiki
Canvas3d	•	1.2.4	✔	✔	✔	✔	High-level OpenGL widget, http://3dcanvas.tcl-lang.org
cawt		2.9	-	✔	-	-	Paul Obermeier's COM Automation With Tcl package, http://www.cawt.tcl3d.org
ceptcl	•	0.4			✔	✔	stu's "Communications Endpoints for Tcl" incl. UDP, IPv6, http://www3.bell.net/stwo/software/index.html
ck	•	8.x		✔	✔	✔	Curses Tcl Toolkit inspired by Tk, http://www.ch-werner.de/ck
csp		0.1.0	✔	✔	✔	✔	Golang inspired concurrency library for Tcl, https://github.com/securitykiss-com/csp
dbif		2.2	-	-	✔	✔	DBus introspection interface, http://dbus-tcl.sourceforge.net
dbus	•	4.0	-	-	✔	✔	DBus bindings for Tcl, http://dbus-tcl.sourceforge.net
dde	•	1.4	-	✔	-	-	Win32 Dynamic Data Exchange, part of the Tcl core
DiffUtil	•	0.4.2		✔	✔	✔	Peter Spjuth's DiffUtilTcl package, https://github.com/pspjuth/DiffUtilTcl
dmtx	•	0.7.5	✔	✔	✔	✔	Data matrix decoder, http://sourceforge.net/projects/libdmtx, see dmtx command
espeak		0.2		✔	✔	✔	Tcl interface to the espeak/espeak-ng library for speech output using Ffidl and TclOO, part of undroidwish
Expect	•	5.45.4	✔	-			Automation for interactive programs, http://expect.sourceforge.net
Ffidl	•	0.7	✔	✔	✔	✔	Foreign function interface with dynamic loading using libffi, https://github.com/prs-de/ffidl
flexmenu		1.52	✔	✔	✔	✔	Brad Lanam's flexmenu alternative menu system, see https://sourceforge.net/projects/tcl-flexmenu
fsdialog		1.15	✔	✔	✔	✔	Schelte Bron's ttk file selection dialog, http://chiselapp.com/user/schelte/repository/fsdialog
fswatch	•	2.0.1	✔	-	-	✔	File system watcher based on inotify, http://chiselapp.com/user/schelte/repository/fswatch
fuse	•	1.1		-	✔	✔	Tcl interface to the linux kernel's FUSE subsystem, https://sourceforge.net/projects/tcl-fuse
gridplus		2.11	✔	✔	✔	✔	Grid based layout system, http://www.satisoft.com/tcltk/gridplus2
helpviewer		3.0.2	✔	✔	✔	✔	Johann Oberdorfer's helpviewer using TkHTML3, http://www.johann-oberdorfer.eu/blog/2017/04/10/17-10-04_helpviewer
icons		2.0	✔	✔	✔	✔	Icon sets, http://www.satisoft.com/tcltk/icons
Img	•	1.4.11	✔	✔	✔	✔	Support for many image formats, http://sourceforge.net/projects/tkimg

imgjp2	•	0.1	✔	✔	✔	✔	JP2 image format handler based on OpenJPEG
iocp	•	1.1.0	-	✔	-	-	Fast inet and bluetooth sockets for Windows by Ashok P. Nadkarni, see https://iocp.magicsplat.com
itcl	•	4.2.0	✔	✔	✔	✔	Tcl object system, http://core.tcl-lang.org/itcl
itk	•	4.1.0	✔	✔	✔	✔	Framework for mega widgets based on itcl, http://core.tcl-lang.org/itk
iwidgets		4.1	✔	✔	✔	✔	Object oriented mega widgets based on itk, http://core.tcl-lang.org/iwidgets
kafka	•	2.4.3		✔	✔	✔	FlightAware's Tcl interface to the Apache Kafka distributed messaging system, https://github.com/flightaware/kafkatcl
materialicons		0.2	✔	✔	✔	✔	Package wrapping the Material Design Icons, part of AndroWish
Memchan	•	2.4	✔	✔	✔	✔	Memory channels, http://memchan.sourceforge.net
modbus		0.1	✔	✔	✔	✔	Tcl modbus interface (see http://libmodbus.org) using Ffidl and TclOO.
Mpexpr	•	1.2	✔	✔	✔	✔	Multi precision math package, https://core.tcl-lang.org/mpexpr
mqtt		3.1	✔	✔	✔	✔	MQTT library including simple broker by Schelte Bron, https://chiselapp.com/user/schelte/repository/mqtt
msgpack		2.0	✔	✔	✔	✔	A pure Tcl implementation of the MessagePack object serialization library, https://github.com/jdc8/msgpack
muzic	•	1.0	✔	-	-	-	MIDI sound package, part of AndroWish, see Muzic MIDI sound package
nats		3.0	✔	✔	✔	✔	Tcl client library for the NATS message broker, https://github.com/Kazmirchuk/nats-tcl
notebook		2.2.0	✔	✔	✔	✔	Will Duquette's notebook app, https://github.com/wduquette/notebook
nsf	•	2.4.0	✔	✔	✔	✔	New Scripting Framework, http://next-scripting.org
ooxml		1.7	✔	✔	✔	✔	Read and write Office Open XML "XLSX" since Excel 2007, https://tcl.sowaswie.de/repos/fossil/ooxml
parse_args	•	0.5.1	✔	✔	✔	✔	A fast argument parser based on the patterns established by core Tcl commands, https://github.com/RubyLane/parse_args
parser	•	1.8	✔	✔	✔	✔	Tcl parser component, https://chiselapp.com/user/aspect/repository/tclparser
pdf4tcl		0.9.4	✔	✔	✔	✔	PDF document generation, http://sourceforge.net/projects/pdf4tcl
pdf4tcl_graph		1.0	✔	✔	✔	✔	BLT/RBC commands for the pdf4tcl library, http://sesam-gmbh.org/images/Downloads/Public/pdf4tcl_graph.zip
piio	•	1.1	-	-	-	✔	Schelte Bron's RaspberryPi GPIO/TWI/SPI library, http://chiselapp.com/user/schelte/repository/piio
pikchr	•	1.0	✔	✔	✔	✔	DRH's pikchr Tcl package, https://pikchr.org
promise		1.1.0	✔	✔	✔	✔	Promise abstraction for asynchronous programming, http://tcl-promise.magicsplat.com
pty	•	0.1	✔	-	✔	✔	Tcl package to handle pseudo TTYs, https://github.com/lawrencewoodman/pty_tcl
ral	•	0.12.2	✔	✔	✔	✔	Relational algebra, http://chiselapp.com/user/mangoa01/repository/tclral
ralutil		0.12.2	✔	✔	✔	✔	Relational algebra, http://chiselapp.com/user/mangoa01/repository/tclral
reg	•	1.3	-	✔	-	-	Win32 Registry, part of the Tcl core
retcl		0.4.0	✔	✔	✔	✔	Redis client library for Tcl, https://gahr.github.io/retcl
rfcomm	•	1.0	✔	-	-		Support for Bluetooth serial port profile, part of AndroWish, see rfcomm command
rl_json	•	0.15.1	✔	✔	✔	✔	JSON value type extension, https://github.com/RubyLane/rl_json
rmq		1.4.5	✔	✔	✔	✔	Pure Tcl Library for RabbitMQ, https://github.com/flightaware/tclrmq
Rtcl	•	1.2.2		✔	✔	✔	Tcl extension embedding "R" Project for Statistical Computing, https://github.com/mattadams/Rtcl
scrolldata		2.12	✔	✔	✔	✔	Virtual Scrolling without a frame or canvas wrapper, https://sourceforge.net/projects/tcl-virtualscrolling
snap7		0.1		✔	✔	✔	Tcl interface to snap7, see http://snap7.sourceforge.net/
snack	•	2.2.10	✔	✔	✔	✔	Sound toolkit (MP3 and OGG support not provided), http://www.speech.kth.se/snack
SOAP		1.6.8	✔	✔	✔	✔	Tcl SOAP interface, http://sourceforge.net/projects/tclsoap
sqlite3	•	3.45.1	✔	✔	✔	✔	Embedded SQL database, http://www.sqlite.org
starDOM		0.42	✔	✔	✔	✔	Small XML browser/editor based on tdom and BWidget, http://wiki.tcl-lang.org/3895
stbimage	•	0.8	✔	✔	✔	✔	Danilo Chang's Tcl binding to stb_image, https://github.com/ray2501/tcl-stbimage
tbcload	•	1.7	✔	✔	✔	✔	Byte-code loader, http://wiki.tcl-lang.org/2624
tcl	•	8.6.10	✔	✔	✔	✔	Tcl core, http://www.tcl-lang.org
tcl-augeas	•	0.4.0	-	-	✔	✔	Tcl binding to augeas, https://github.com/dbohdan/tcl-augeas
tclcan	•	0.1		-	-	✔	Tcl interface to Linux SocketCAN raw AF_CAN sockets, part of undroidwish, see tclcan
tclcsv	•	2.3	✔	✔	✔	✔	The tclcsv extension by Ashok P. Nadkarni, http://tclcsv.magicsplat.com/
tclcompiler	•	1.7.1		✔	✔	✔	Tcl compiler from TDK, https://github.com/andreas-kupries/tdk/

TclCurl	•	7.22.0	✔	✔	✔	✔	Tcl interface to `curl` library, https://github.com/flightaware/tclcurl-fa
tclepeg	•	0.4	✔	✔	✔	✔	Tcl extension to the epeg thumbnailing library, https://github.com/dzach/tclepeg
tcLex	•	1.2		✔	✔	✔	Lexer (lexical analyzer) generator extension to Tcl, https://salsa.debian.org/tcltk-team/tclex
tclhttpd		3.5.3		✔	✔	✔	Tcl based web server, http://tclhttpd.sourceforge.net
tclJBlend	•	2.1.0	✔	✔	✔	✔	Tcl extension using JNI to communicate with a Java VM, https://sourceforge.net/projects/irrational-numbers/files
tcllib		1.21	✔	✔	✔	✔	Tcl standard library, http://core.tcl-lang.org/tcllib
tcl-lmdb	•	0.4.3	✔	✔	✔	✔	Tcl interface to the Lightning Memory-Mapped Database, https://sites.google.com/site/ray2501/tcl-lmdb
TclMagick	•	0.46		✔	✔	✔	Tcl interface to the GraphicsMagick image processing system, http://www.graphicsmagick.org
TclMixer	•	1.2.3	✔				Tcl interface to `SDL2_mixer` (music and sound playback), http://sqlitestudio.pl/tclmixer
tcltaglib	•	1.1				✔	Tcl interface to TagLib audio meta-data library, https://github.com/ray2501/tcltaglib
tcluvc	•	0.1	✔	-	✔	✔	Tcl interface to UVC type cameras based on libuvc and libusb
tclwmf	•	0.1	-	✔	-	-	Tcl interface to cameras using Windows Media Foundation, see wmf command
Tclx	•	8.6	✔	✔	✔	✔	Extended Tcl, https://github.com/flightaware/tclx
tdbc	•	1.1.1	✔	✔	✔	✔	Tcl database connectivity, http://core.tcl-lang.org/tdbc
tdbc::jdbc		0.2.0	✔	✔	✔	✔	TDBC-JDBC bridge, https://github.com/ray2501/TDBCJDBC
tdbc::mysql	•	1.1.1		✔	✔	✔	TDBC driver for MySQL, http://core.tcl-lang.org/tdbcmysql
tdbc::odbc	•	1.1.1		✔	✔	✔	TDBC driver for ODBC, http://core.tcl-lang.org/tdbcodbc
tdbc::postgres	•	1.1.1		✔	✔	✔	TDBC driver for PostgreSQL, http://core.tcl-lang.org/tdbcpostgres
tdbc::sqlite3		1.1.1	✔	✔	✔	✔	TDBC driver for sqlite3, http://core.tcl-lang.org/tdbcsqlite3
TDK	•			✔	✔	✔	subset of Tcl Dev Kit from https://github.com/tcltk/tdk
tdom	•	0.9.3	✔	✔	✔	✔	XML/DOM/XPath/XSLT implementation for Tcl, http://tdom.org/index.html
tfirmata		2.5?	✔	✔	✔	✔	Tcl implementation of Arduino Firmata, https://wiki.tcl-lang.org/page/Firmata
Thread	•	2.8.5	✔	✔	✔	✔	Tcl thread extension, http://core.tcl-lang.org/thread
ticklecharts		3.2.2		✔	✔	✔	Nicolas Robert's tickleEcharts package, https://github.com/nico-robert/ticklecharts
tile-extras		various	✔	✔	✔	✔	Misc. bag of Tk packages related to the Tile widget set, https://github.com/jenglish/tile-extras
Tix	•	8.4.3	✔	✔	✔	✔	Alternate widget set, http://tix.cvs.sourceforge.net/tix/tix
tk	•	8.6.10	✔	✔	✔	✔	Tk toolkit, http://www.tcl-lang.org
tkcon		2.7	✔	✔	✔	✔	Tk console, http://tkcon.sourceforge.net
tkconclient		1.0	✔	✔	✔	✔	Remote support for Tk console, borrowed from Tcl wiki, part of AndroWish
TkDND	•	2.9.2	-	✔	✔	✔	Tk drag and drop interface, https://github.com/petasis/tkdnd
Tkhtml	•	3.0	✔	✔	✔	✔	Tk HTML widget, http://tkhtml.tcl-lang.org.tk
tkinspect		5.1.6	✔	✔	✔	✔	Tool to inspect contents of other running Tk applications, http://sourceforge.net/projects/tkcon/files
tklib		0.7	✔	✔	✔	✔	Tk standard library, http://core.tcl-lang.org/tklib
tkpath	•	0.3.3	✔	✔	✔	✔	Alternate canvas widget with SVG like capabilities, https://bitbucket.org/andrew_shadura/tkpath
tksqlite		0.5.13	✔	✔	✔	✔	GUI frontend to sqlite3, http://reddog.s35.xrea.com/wiki/TkSQLite.html
tksvg	•	0.14	✔	✔	✔	✔	Read SVG to Tk photo images, https://github.com/oehhar/tksvg
Tktable	•	2.11	✔	✔	✔	✔	Tk table widget, http://tktable.sourceforge.net
tkvlc	•	0.8	-	✔	✔	✔	Video playback using libVLC, https://github.com/ray2501/tkvlc
tktray	•	1.3.9	-	-	-	✔	Manage system tray icons with Tk on X11, http://code.google.com/p/tktray
Tkzinc	•	3.3.6	✔	✔	✔	✔	TkZinc widget, similar to Tk's canvas, https://bitbucket.org/plecoanet/tkzinc
tls	•	1.6	✔	✔	✔	✔	Tcl interface to OpenSSL/LibreSSL, http://tls.sourceforge.net
tomato		1.2.3	✔	✔	✔	✔	Nicolas Robert's tomato math::geometry 3D library, https://github.com/nico-robert/tomato
topcua	•	0.5	✔	✔	✔	✔	Proof of concept Tcl binding to https://open62541.org, part of AndroWish
treectrl	•	2.4.2	✔	✔	✔	✔	Tk tree widget, http://sourceforge.net/projects/tktreectrl
Trf	•	2.1.4	✔	✔	✔	✔	Transformation procedure framework for Tcl channels, http://tcltrf.sourceforge.net
trofs	•	0.4.9	✔	✔	✔	✔	Tcl read-only filesystem, http://math.nist.gov/~DPorter/tcltk/trofs
tserialport	•	1.1		✔	✔	✔	Alexander Schoepe's extension to query serial ports, https://tcl.sowaswie.de/tserialport

TWAPI	•	4.7.2	-	✔	-	-	Tcl Windows API extension, http://twapi.magicsplat.com
twv	•	0.1	-	✔	✔	✔	Simple Tcl Webview (WIP), part of undroidwish
udp	•	1.0.11	✔	✔	✔	✔	UDP sockets, http://core.tcl-lang.org/tcludp
ukaz		2.1	✔	✔	✔	✔	Graph widget written in pure Tcl/Tk, http://github.com/auriocus/ukaz
unqlite	•	0.3.8		✔	✔	✔	Tcl interface to the UnQLite library, https://github.com/ray2501/tclunqlite
upnp		0.2	✔	✔	✔	✔	Universal Plug and Play, http://chiselapp.com/user/schelte/repository/upnp
usbserial	•	1.0	✔	-	-	-	Support for USB serial converters, part of AndroWish, see usbserial command
v4l2	•	0.1	-	-	-	✔	Video For Linux Two interface, see v4l2 command
vcd		0.1	✔	✔	✔	✔	Trace facility using Verilog VCD (Value Change Dump) format, see https://wiki.tcl-lang.org/page/VCD
VecTcl	•	0.3	✔	✔	✔	✔	Numerical math in Tcl, http://auriocus.github.io/VecTcl
VecTcLab			✔	✔	✔	✔	Console for VecTcl derived from tkcon, http://github.com/auriocus/VecTcLab
vfs	•	1.4.2	✔	✔	✔	✔	Virtual file system in Tcl, https://core.tcl-lang.org/tclvfs/index
vnc	•	0.5	✔	✔	✔	✔	VNC viewer widget, http://ch-werner.de/tkvnc
vlerq	•	4.1	✔	✔	✔	✔	Package for managing structured datasets in Tcl, https://web.archive.org/web/20161012011244/http://equi4.com/vlerq.org/
vu	•	2.3	✔	✔	✔	✔	Various Tk widgets, http://tktable.sf.net
wibble		0.4	✔	✔	✔	✔	Small web server, http://chiselapp.com/user/andy/repository/wibble
WiTS		3.2.5	-	✔	-	-	Windows Inspection Tool Set, http://windowstoolset.sourceforge.net
WS		2.7.1	✔	✔	✔	✔	Tcl interface to web services, http://core.tcl-lang.org/tclws
xml	•	3.2	✔				Tcl interface to XML, http://sf.net/projects/tclxml
winhelp	•	1.1	-	✔	-	-	Tcl interface to Windows HTML Help, http://www.ch-werner.de/winhelp
winsend	•	1.0	-	✔	-	-	Tk send command under windows using COM, https://sourceforge.net/projects/tclsoap/files/winsend
www		2.4	✔	✔	✔	✔	Schelte Bron's www package, https://chiselapp.com/user/schelte/repository/www
yeti		0.4.2	✔	✔	✔	✔	Generate an itcl parser for a BNF-like grammar, http://www.fpx.de/fp/Software/Yeti
zbar	•	0.10	✔	✔	✔	✔	Barcode scanner, http://zbar.sourceforge.net, see zbar command
zint	•	2.13.0	✔	✔	✔	✔	Barcode generation, http://sourceforge.net/projects/zint

Beyond AndroWish

Some subdirectories of AndroWish have a ready-to-build-then-use Debian infrastructure built in. That allows to build Debian packages easily e.g. on a Raspbian distribution running on your Raspberry Pi:

```
cd .../jni/SDL2 ; dpkg-buildpackage -tc -uc ; dpkg -i ../libsdl2*.deb
cd .../jni/tcl ; dpkg-buildpackage -tc -uc ; dpkg -i ../sdltcl*.deb
cd .../jni/sdl2tk ; dpkg-buildpackage -tc -uc ; dpkg -i ../sdl2tk*.deb
/opt/sdltk86/bin/sdl2wish8.6
```

Building some components of AndroWish for the Windows OS family is possible, too, by using cross compilation on a Linux system. More information can be found in undroidwish.

The resulting `sdl2wish8.6`, `sdl2wish86.exe`, or undroidwish binaries support additional command line options to control certain SDL features. Important: these options must be specified on the command line after the optional script to be executed:

`-sdlfullscreen`

Make the SDL window (the root window for Tk) into a fullscreen window.

`-sdlresizable`

Allow resizing of the SDL window.

`-sdlnoborder`

Make the SDL window borderless, i.e. without window manager decorations.

`-sdlheight` *`pixels`*

Set the height of the SDL window to *`pixels`*.

`-sdlwidth` *`pixels`*

Set the width of the SDL window to *`pixels`*.

`-sdlrootheight` *`pixels`*

Set the height of the root window (as seen by Tk) to *`pixels`*. If not set, the root window's size is equal to the SDL window size.

`-sdlrootwidth` *`pixels`*

Set the width of the root window (as seen by Tk) to *`pixels`*. If not set, the root window's size is equal to the SDL window size.

`-sdlxdpi` *`dpi`*

Set the dots per inch ratio for the X dimension to *`dpi`*. If both, `-sdlxdpi` and `-sdlydpi` are not set, the default is approx. 75 dpi. If only one dimension is set (`-sdlxdpi` or `-sdlydpi`), that value is taken as overall dots per inch ratio.

`-sdlydpi` *`dpi`*

Set the dots per inch ratio for the Y dimension to *`dpi`*. If both, `-sdlxdpi` and `-sdlydpi` are not set, the default is approx. 75 dpi. If only one dimension is set (`-sdlxdpi` or `-sdlydpi`), that value is taken as overall dots per inch ratio.

`-sdlnogl`

Force using the software renderer. This turns OpenGL usage off.

`-sdllog` *`level`*

Set the minimum log level to be shown in SDL log message. *`level`* must be a positive integer.

`-sdlicon` *`filename`*

Set the SDL root window icon to the BMP image from *`filename`*.

`-sdlnosysfonts`

Don't search for and register system fonts. This can reduce startup time significantly.

`-sdlopacity` *`value`*

Set the initial opacity of the SDL root window. *`value`* must be given as positive integer percentage.

`-sdlswcursor`

Force use of a software cursor texture. Useful, when no proper hardware cursor support is available, e.g. in Haiku using the OpenGL render driver.

Some SDL runtime switches must be specified early by setting environment variables. All these switches are documented in the **SDL_hints.h** header file. The most important are:

`SDL_VIDEODRIVER`

A string selecting the video driver, use it to enable the **jsmpeg** video driver.

`SDL_RENDER_DRIVER`

A string selecting the SDL renderer. Normally chosen automatically but sometimes it can be necessary to explicitly turn on the **software** renderer. Other possible values depend on how SDL was built, e.g **opengl**, **opengles2** etc.

ble command

ble command

Name

ble - interact with Bluetooth Low Energy (BLE) devices. Requires Android 4.3 or higher.

Synopsis

```
package require Ble
ble subcommand ?options?
```

Description

This command is used to deal with Bluetooth Low Energy (BLE) devices. The legal `subcommands` (which may be abbreviated) are:

`ble abort` *`handle`*

> Abort the current write transaction on the BLE connection identified by *`handle`* which was obtained earlier by a `ble connect` command. Returns an integer indicating success (1), failure (0), or system error (less than 0).

`ble begin` *`handle`*

> Starts a write transaction on the BLE connection identified by *`handle`* which was obtained earlier by a `ble connect` command. Returns an integer indicating success (1), failure (0), or system error (less than 0).

`ble callback` *`handle ?callback?`*

> If the *`callback`* argument is provided that argument replaces the callback function on the BLE connection identified by *`handle`* and returns the old callback function. Otherwise the current callback function is returned. In contrast to e.g. the Tk event bind mechanism, the *`callback`* argument has not all the freedom of a Tcl bind script, i.e. it must be a single command and be parseable as a list since internally the Tcl core function Tcl_EvalObjv() is used for executing the callback instead of the Tcl_Eval*() function family supporting full scripts.

`ble characteristics` *`handle suuid sinstance`*

> Returns a list of characteristics of the service described by its UUID *`suuid`* and instance number *`sinstance`* on the BLE connection *`handle`*. The list is layed out as a table with the five columns characteristic UUID, characteristic instance number, permissions, properties, and write type suitable for iterating using `foreach {cuuid cinstance perm prop wrtype} [ble characteristics ...] {...}`.

`ble close` *`handle`*

> Closes the BLE connection identified by *`handle`* which was obtained earlier by a `ble connect` or `ble scanner` command.

`ble connect` *`address callback ?flag?`*

> Connects to the Bluetooth LE device with address *`address`* (expressed as six hexadecimal 8 bit numbers separated by colons, like a Ethernet MAC address), and arranges for the *`callback`* command to be invoked on events on the connection to this device. The optional *`flag`* is a boolean with default false controlling automatic connection setup (see the Android documentation for more details). The callback command is called with two additional arguments, the first is a string (`connection`, `scan`, `service`, `characteristic`, `descriptor`, or `transaction`) indicating the kind of event, the second is a dictionary with event related information, see the section **Event Data** below. For restrictions of the *`callback`* argument see the description in `ble callback` above. The result of the `ble connect` command is a handle (a string identifying the BLE connection) to be used in other `ble` subcommands. During connection establishment an automatic discovery takes place which detects all advertised services, characteristics, and descriptors of the remote Bluetooth LE device.

`ble descriptors` *`handle suuid sinstance cuuid cinstance`*

> Returns a list of descriptors of the service and characteristic described by its UUIDs *`suuid`* and *`cuuid`* and instance numbers *`sinstance`* and *`cinstance`* on the BLE connection *`handle`*. The list is layed out as a table with the two columns descriptor UUID and permissions suitable for iterating using `foreach {duuid perm} [ble descriptors ...] {...}`.

`ble disable` *`handle suuid sinstance cuuid cinstance`*

> Turns off notifications of a characteristic of the BLE connection *`handle`* identified by *`suuid`* (128 bit service UUID), *`sinstance`* (service instance identifier, integer, usually 0), *`cuuid`* (128 bit characteristic UUID), and *`cinstance`* (characteristic identifier, integer, usually 0).

`ble disconnect` *`handle`*

Initiates a disconnect of the BLE connection *`handle`* if the current connection state is disconnected. When the operation completes the callback function of the connection is invoked.

`ble dread` *`handle suuid sinstance cuuid cinstance duuid`*

Initiates the read of a descriptor of the BLE connection *`handle`* identified by *`suuid`* (128 bit service UUID), *`sinstance`* (service instance identifier, integer, usually 0), *`cuuid`* (128 bit characteristic UUID), *`cinstance`* (characteristic identifier, integer, usually 0), and *`duuid`* (128 bit descriptor UUID). The result is a positive integer when the descriptor read operation is in progress, 0 or negative on error. The completion of the descriptor read operation is indicated through the callback function of the connection.

`ble dwrite` *`handle suuid sinstance cuuid cinstance duuid value`*

Initiates the write of a descriptor of the BLE connection *`handle`* identified by *`suuid`* (128 bit service UUID), *`sinstance`* (service instance identifier, integer, usually 0), *`cuuid`* (128 bit characteristic UUID), *`cinstance`* (characteristic identifier, integer, usually 0), and *`duuid`* (128 bit descriptor UUID). *`value`* is the value to be written and should be a string or byte array. The result is a positive integer when the descriptor write operation is in progress, 0 or negative on error. The completion of the descriptor write operation is indicated through the callback function of the connection.

`ble enable` *`handle suuid sinstance cuuid cinstance`*

Turns on notifications of a characteristic of the BLE connection *`handle`* identified by *`suuid`* (128 bit service UUID), *`sinstance`* (service instance identifier, integer, usually 0), *`cuuid`* (128 bit characteristic UUID), and *`cinstance`* (characteristic identifier, integer, usually 0).

`ble equal` *`handle uuid1 uuid2`*

Tests if the given UUIDs are equal. Both can be specified in abbreviated form and are expanded before comparison. Returns true, if the UUIDs are the same. Unknown abbreviated or long UUIDs with respect to the connection *`handle`* compare always to false.

`ble execute` *`handle`*

Dispatches execute (commit) of the current write transaction which was started earlier using `ble begin` on the BLE connection identified by *`handle`* which was obtained earlier by a `ble connect` command. Returns an integer indicating success (1), failure (0), or system error (less than 0). The result of the transaction is reported in the callback with event kind `transaction`.

`ble expand` *`handle uuid`*

Expands the given (abbreviated, short) UUID to its 128 bit (long, canonical) form and returns a 128 bit UUID string. An error is reported if an abbreviated or long UUID is unknown with respect to the connection *`handle`*.

`ble getrssi` *`handle`*

Requests remote SSI information from the BLE connection identified by *`handle`* which was obtained earlier by a `ble connect` command. Returns an integer indicating success (1), failure (0), or system error (less than 0). The updated remote SSI is reported in later callbacks.

`ble info` *`?handle?`*

Returns information of the BLE connection identified by `handle` as a dictionary made up the fields *handle* (the connection identifier), *address* (Bluetooth address), and *state* (connection state, one of *disconnected*, *discovery*, *scanning*, *connected*, or *idle*. If *`handle`* is omitted, a list of all known connection identifiers is returned.

`ble mtu` *`handle ?value?`*

Returns or sets the maximum transmission unit (MTU) of the BLE connection identified by *`handle`*. Support of this function varies between Android versions.

`ble pair` *`address`*

Initiates pairing with the Bluetooth device with address *`address`* (expressed as six hexadecimal 8 bit numbers separated by colons, like a Ethernet MAC address).

`ble read` *`handle suuid sinstance cuuid cinstance`*

Initiates the read of a characteristic of the BLE connection *`handle`* identified by *`suuid`* (128 bit service UUID), *`sinstance`* (service instance identifier, integer, usually 0), *`cuuid`* (128 bit characteristic UUID), and *`cinstance`* (characteristic identifier, integer, usually 0). The result is a positive integer when the read operation is in progress, 0 or negative on error. The completion of the read operation is indicated through the callback function of the connection.

`ble reconnect` *`handle`*

Initiates a reconnect of the BLE connection *`handle`* if the current connection state is disconnected. When

the operation completes the callback function of the connection is invoked with information on the new connection state.

`ble scanner` *`callback`*

Creates a BLE connection to be used for detection (scan) of BLE devices and returns a handle (a string identifying the BLE scanner) to be used to deal with this scanner and arranges for the *`callback`* command to be invoked on events on the connection. See the description of `ble connect` and the section **Event Data** for more details on the `callback` argument.

`ble services` *`handle`*

Returns a list of services of the BLE connection *`handle`*. The list is layed out as a table with the three columns service UUID, service instance number, and service type suitable for iterating using `foreach {suuid sinstance type} [ble services ...] {...}`.

`ble shorten` *`handle uuid`*

Shorten the given (long, canonical) UUID to its shortest (16 or 32 bit) form. An error is reported if the long UUID is unknown with respect to the connection *`handle`*. If no unique abbreviation is found, the full 128 bit UUID is returned.

`ble start` *`handle`*

Starts scanning for BLE devices. Scan status and scan results are indicated by invocations of the *`callback`* function given to the corresponding `ble scanner` command.

`ble stop` *`handle`*

Stops scanning for BLE devices. Scan status is indicated by invocations of the *`callback`* function given to the corresponding `ble scanner` command.

`ble unpair` *`address`*

Initiates release of the pairing with the Bluetooth device with address *`address`* (expressed as six hexadecimal 8 bit numbers separated by colons, like a Ethernet MAC address).

`ble userdata` *`handle ?data?`*

Associate or retrieve user data with the BLE connection *`handle`*. When *`data`* is given it replaces the former associated user data. When omitted, the current user data or an empty list is returned.

`ble write` *`handle suuid sinstance cuuid cinstance value`*

Initiates the write of a characteristic of the BLE connection *`handle`* identified by *`suuid`* (128 bit service UUID), *`sinstance`* (service instance identifier, integer, usually 0), *`cuuid`* (128 bit characteristic UUID), and *`cinstance`* (characteristic identifier, integer, usually 0). *`value`* is the value to be written and should be a string or byte array. The result is a positive integer when the write operation is in progress, 0 or negative on error. The completion of the write operation is indicated through the callback function of the connection. Note: not all Android implementations allow more than one active command (example, issuing a "ble read" immediately after a "ble write"). For best compatibility, you should wait for the callback that your write operation has completed before issuing the next ble write/read command.

Abbreviated UUIDs

The 128 bit UUID arguments to `ble` commands can be specified in abbreviated 16 or 32 bit form as long as the value is unique with respect to the UUIDs learned during the discovery phase. Examples:

```
TI SensorTag Base UUID:        F0000000-0451-4000-B000-000000000000

IR Temperature Sensor Service: F000AA00-0451-4000-B000-000000000000
         abbreviated (32 bit): F000AA00
         abbreviated (16 bit):     AA00

IR Temperature Sensor Value:   F000AA01-0451-4000-B000-000000000000
         abbreviated (32 bit): F000AA01
         abbreviated (16 bit):     AA01

Generic descriptor for notify: 00002902-0000-1000-8000-00805F9B34FB
         abbreviated (32 bit): 00002902
         abbreviated (16 bit):     2902
```

Event Data

The first argument to callback functions is the type of event, as described below.

`connection`

Indicates change in connection state.

`scan`

Indicates change in scan state or reports newly detected Bluetooth LE devices.

`service`

Information about a service.

`characteristic`

Information about a characteristic, used for data exchange.

`descriptor`

Information about a descriptor (meta information of a characteristic).

`transaction`

Indicates status of a write transaction.

The second argument to callback functions is a dictionary with keys depending on the kind of the event. The following paragraphs list the various event formats.

`handle` *`h`* `state` *`s`*

Connection state event for `ble scan`. `state` can be one of `scanning` or `idle`.

`handle` *`h`* `address` *`a`* `state` *`s`* `rssi` *`r`*

Connection state event for `ble connect`. `state` can be one of `disconnected`, `discovery`, or `connected`. In the discovery phase the services, characteristics, and descriptors of the remote device are gathered. The `rssi` field contains the last read remote SSI (signal strength indicator) in dBm as integer number.

`handle` *`h`* `state` *`s`* `address` *`a`* `name` *`n`* `type` *`t`* `rssi` *`r`*

Scan result event. `address` is the Bluetooth address of the remote device, `name` it's advertised friendly name, `type` the device type as integer, `rssi` the receive SSI in dBm as integer.

`handle` *`h`* `address` *`a`* `state` *`s`* `rssi` *`r`* `suuid` *`su`* `sinstance` *`si`* `type` *`t`*

Service discovery event. `suuid` is the UUID of the service, `sinstance` the instance of that service as integer number. Refer to BluetoothGattService for details.

`handle` *`h`* `address` *`a`* `state` *`s`* `rssi` *`r`* `suuid` *`su`* `sinstance` *`si`* `cuuid` *`ci`* `cinstance` *`ci`* `permissions` *`p`* `properties` *`q`* `writetype` *`w`* `access` *`a`* `value` *`v`*

Characteristic event. `cuuid` is the UUID of the characteristic, `cinstance` the instance of that characteristic as integer number. The items `permission`, `properties`, and `writetype` are integer numbers, too. The `access` item contains a one letter code indicating the type of access ('c' for change notification, 'd' for discovery, 'r' for read, 'w' for write). The `value` item holds the data of the characteristic as a byte array. It's interpretation is device/characteristic depending. This event is reported during discovery and normal operation when `ble read` or `ble write` are performed locally or notifications for the characteristic are enabled using `ble enable`. Refer to BluetoothGattCharacteristic for details.

`handle` *`h`* `address` *`a`* `state` *`s`* `rssi` *`r`* `suuid` *`su`* `sinstance` *`si`* `cuuid` *`ci`* `cinstance` *`ci`* `duuid` *`di`* `permissions` *`p`* `access` *`a`* `value` *`v`*

Descriptor event. `duuid` is the UUID of the descriptor. The item `permission` is an integer number, too. The `access` item contains a one letter code indicating the type of access ('d' for discovery, 'r' for read, 'w' for write). The `value` item holds the data of the descriptor as a byte array. It's interpretation is device/characteristic/descriptor depending. This event is reported during discovery and normal operation when `ble dread` or `ble dwrite` are performed locally. Refer to BluetoothGattDescriptor for details.

`handle` *`h`* `success` *`s`*

Transaction result event. `success` is the transaction result and is 1 for success or 0 for failure.

Example

The following example scans for Bluetooth LE devices, connects to a TI SensorTag and enables notifications of the buttons of the device.

```
proc ble_handler {what data} {
    switch -- $what {
        scan {
            if {[dict get $data name] eq "SensorTag"} {
                # found the TI SensorTag, connect it, stop the scanner
                ble connect [dict get $data address] ble_handler 1
                ble close [dict get $data handle]
            }
        }
        connection {
            if {[dict get $data state] == "disconnected"} {
```

```
                # fall back to scanning
                ble close [dict get $data handle]
                ble start [ble scanner ble_handler]
            } elseif {[dict get $data state] == "connected"} {
                # if the TI SensorTag buttons were found,
                # it will be enabled for notifications now
                set handle [dict get $data handle]
                set cmd [ble userdata $handle]
                if {$cmd ne {}} {
                    if {[{*}$cmd]} {
                        # success, clear userdata
                        ble userdata $handle {}
                    }
                }
            }
        }
        descriptor {
            if {[string match "*2902-*" [dict get $data duuid]] &&
                [string match "*FFE1-*" [dict get $data cuuid]]} {
                # descriptor for TI SensorTag buttons found
                set flag 0
                # notification enable state, 16 bit little-endian
                #  0x0000 = disabled, 0x0001 = enabled
                binary scan [dict get $data value] s flag
                if {!$flag} {
                    # later turn on notifications of button changes
                    set handle [dict get $data handle]
                    ble userdata $handle [list ble enable $handle \
                        [dict get $data suuid] [dict get $data sinstance] \
                        [dict get $data cuuid] [dict get $data cinstance]]
                }
            }
        }
    }
    # dump data to stdout
    if {[dict exists $data value]} {
        # make hex string from byte array
        binary scan [dict get $data value] H* value
        dict set data value $value
    }
    puts "$what: $data"
}

ble start [ble scanner ble_handler]
```

Build custom Androwish

Starting point

Starting point is the description by Christian at http://www.androwish.org/index.html/wiki?name=Building+AndroWish and the following quote from wiki page Androwish:

Please fetch the sources (the big .tar.bz2), unpack it, have Android SDK and NDK installed, don't use Eclipse, adapt local.properties to where you've installed Android SDK, have your PATH properly set so that ndk-build can do its job, then invoke "ant debug", be patient, and you'll finally will have bin/AndroWish-debug.apk ready to be installed onto your device. I have never verified the build process in combination with Eclipse. Once upon a time, I did my very first steps using the tips from the SDL documentation regarding Android.

When you want to wrap your own app written as Tcl code, you should add it below assets/app, have the launching script as main.tcl, fiddle the toplevel AndroidManifest.xml to have your app/class name in, remove that AndroWishScript/Launcher stuff from the manifest (since not needed for a standalone app), derive your app main class (yes, some Java required) from src/tk/tcl/wish/AndroWish.java, e.g.

```
import tk.tcl.wish.AndroWish;
public class TclTkRules extends AndroWish {}
```

fiddle the res directory with a new really koooool icon and title for your app.

Build Androwish

Get Source

A release source is on the web site. If an intermediate version should be used, one may clone the fossil repository and check out the latest checkin on trunk:

```
fossil clone http://anonymous:F4DC0163@www.androwish.org androwish.fossil
mkdir androwish
cd androwish
fossil open ../androwish.fossil
rm .fslckout
```

Try on Windows

Windows build stopped with ndk-build with a "command line to long" error. I tried cmd.exe and cygwin shell, same result.

This should be fixed since check-in [52a07071b99fa88a] and was verified on Windows 8.1 32 bit using Android NDK r12b and Android SDK 24.4.1.

Try on OSX

NDK: I downloaded android-ndk-r10e-darwin-x86_64.bin - then chmod +x, execute it, and move extracted files to /usr/local/android-ndk

Added this to ~/.bash_profile:

```
export NDK_PROJECT_PATH=/usr/local/android-ndk
export ANDROID_HOME=~/Library/Android/sdk
export PATH=${PATH}:~/android-sdk-mac/tools:/usr/local/android-ndk
```

edited "project.properties" to update the android target number.

```
# Project target.
target=android-21
```

To build:

```
cd ~/Documents/androwish
export NDK_PROJECT_PATH=`pwd`
ant debug
```

"ant debug" runs for about 30 minutes, and ends with

```
BUILD SUCCESSFUL
```

Try on CentOS 6

Failed for me due to a to old clib.

Christian: remarked that he is using CentOS 6 or Ubuntu 12.04 LTS with Andriod NDK 9d. So this failure might be due to the fact, that I tried Android NDK 10d.

OpenSuSE 13.2 64 bit

I installed VirtualBox on my Windows 8.1 and OpenSuSE 13.2 64 bit with 100GB HarDisk and 4GB Ram.

- Added series: java development
- Added packages: java-1_7_0-openjdk-devel, xerces-j2-xml-apis

Activate Java 7 (e.g. 1.7):

```
update-alternatives --config java
-> 1.7
update-alternatives --config javac
-> 1.7
update-alternatives --config xml-commons-apis
-> xerces-j2-xml-apis.jar
```

Set up Android build system:

```
cd ~
mkdir android
cd android
mkdir download
```

Downloaded in ~/android/download:

- android-sdk_r24.1.2-linux.tgz
- android-ndk-r9d-linux-x86_64.tar.gz
- androwish-e2aee3ea2ea718e7.tar.gz (Pi Day Release, also tested with following Don Quixote Release)

Christian: suggested to use the 9d release of the ndk instead of the current 10d due to the following reasons:

- still supports Android 2.3.3, like AndroWish
- tiff library does not compile with 10d

The download link is:

- Linux 64 bit: https://dl.google.com/android/ndk/android-ndk-r9d-linux-x86_64.tar.bz2
- Linux 32 bit: https://dl.google.com/android/ndk/android-ndk-r9d-linux-x86.tar.bz2

Unpack and install, androwish in folder "androwish" for easier access

```
cd ~/android
tar xvzf download/android-sdk_r24.1.2-linux.tgz
bzip2 -d download/android-ndk-r9d-linux-x86_64.tar.bz2
tar xvf download/android-ndk-r9d-linux-x86_64.tar
tar xvf download/androwish-e2aee3ea2ea718e7.tar.gz
mv androwish-e2aee3ea2ea718e7 androwish
```

(start side note)

---

ndk 10d install instructions (if 9d is not used as above)

```
cd ~/android
chmod +x download/android-ndk-r10d-linux-x86_64.bin
download/android-ndk-r10d-linux-x86_64.bin
```

(end side note)

Open Android SDK manager:

```
~/android/android-sdk-linux/tools/android sdk
-> Select Google APIs ARM EABI v7a System Image
-> Unselect all other system images
```

Prepare build and let "android" create "local.properties":

```
export PATH=$PATH:~/android/android-sdk-linux/tools:~/android/android-ndk-r9d
cd androwish

android update project -p . --target 1
```

(the export command may be copied to ~/.bashrc to be active for each shell start)

(start of side note)

Error with ndk 10d and not with 9d (e.g. only when 10d is used)

On "ant debug", I had the following build error I could not solve:

```
     [exec] [armeabi] Compile thumb  : tiff_tkimg <= tif_predict.c
     [exec] /tmp/ccTUdnr3.s: Assembler messages:
     [exec] /tmp/ccTUdnr3.s: Error: unaligned opcodes detected in executable segment
     [exec] make: [obj/local/armeabi/objs/tiff_tkimg/libtiff/tif_predict.o] Error 1
```

This is in jni/tiff. So I deleted the tkimg and jni/tiff folders:

```
rm -rf jni/tkimg jni/tiff
```

The build error most likely is caused by a compiler problem. This issue is fixed in an AndroWish check-in on 2015-06-30 by compiling libtiff to ARM instead of Thumb code. The x86 version of the compiler does not cause build errors.

(end side note)

Now, an "ant debug" succeeds for me. The result is in "androwish/bin/AndroWish-debug.apk

Great, thank you, Cristian !

Customizing Androwish

This is a customisation for the application called "HIBIScan" for the company url "elmicron.de". You should replace those names by your own ones.

Delete not required packages

It is perhaps me, but I always try to get small packages with as less as possible included. So I deleted packages I know and I don't need in this project:

```
cd jni
rm -rf 3dcanvas blt curl expect itk jpeg libxml2 nsf TclCurl tclral tcludp tclx tclxml\
    tdom tiff Tix tkimg tktable tktreectrl vu xotcl zint
cd jni/tcl-pkgs
rm -rf tdbcmysql1.0.3 tdbcsqlite3-1.0.3 itcl4.0.3 sqlite3.8.8.3 tdbcodbc1.0.3\
    thread2.7.2 tdbc1.0.3 tdbcpostgres1.0.3

cd androwish
rm -rf tkchat

cd assets
rm -rf bin blt2.4 bwidget1.9.7 Canvas3d1.2.1 expect5.45.2 gridplus2.10 icons1.2 itcl4.0.3\
  itk4.0.1 iwidgets4.1 nsf2.0.0 pdf4tcl08 ral0.11.2 ralutil0.11.2 sqlite3 TclCurl7.22.0\
  tcllib1.16 tclsoap1.6.8 tclws2.3.8 tclx8.4.1 Tclxml3.2 tdbc1.0.3 tdbcsqlite3-1.0.3 tdom0.8\
  thread2.7.2 tkimg1.4.3, tklib0.6, tksqlite0.5.11, tktable2.11 treectrl2.4.1 vu2.3
```

This results in an androwish size of 17MB, so 6 MB less than the full package.

Remove target x86

For most Android phones, the target armeabi is sufficient. So the target x86 might be deleted: Remove "x86" in file jni/Application.mk to get:

```
APP_ABI := armeabi
```

This results in a final apk size of 10MB. My phone says that it takes 19.7 MB, while AndroWish takes 39.3MB.

Include own script

Now, the script tree of the application is copied to assets/app and a main.tcl is there to be started:

```
cd assets
mkdir app
cd app
cp <somewhere>/main.tcl .
cp -r <somewhere>/* .
```

An "ant debug" results in a starkit-like apk file.

Remove permissions not required for the app

In "./AndroidManifest.xml", you may delete any permission, but:

```
  <uses-permission android:name="android.permission.INTERNET" />
```

Application will directly terminate if not present.

Change package name

In "./AndroidManifest.xml", you should change the package name to be different to androwish. Otherwise, the applications may not be installed together.

In "./AndroidManifest.xml"

```
          package="de.elmicron.hibiscan"
```

where "de.elmicron.hibiscan" is my internet domain and the application name as last component. This should be adopted on request.

Add into "src/tk/tcl/wish/AndroWish.java" at the end of the include list:

```
import de.elmicron.hibiscan.R;
```

to avoid error:

```
none
    [javac] /home/oehhar/android/androwish-hibiscan/src/tk/tcl/wish/AndroWish.java:1519: error: package
R does not exist
    [javac]                                                 R.drawable.wish);
```

This error only happens after an

```
ant clean
```

Otherwise, the old class definition of "tk.tcl.wish.R" is still present in the gen source tree.

Add a derived class in "src/de/elmicron/hibiscan/HIBIScan.java". The file path is composed of "src" and the package name, dots replaced by "/". The file name is the class name, where I used the application name.

File contents:

```
package de.elmicron.hibiscan;
import tk.tcl.wish.*;
public class HIBIScan extends AndroWish { }
```

(Christian: by private email) Then, each usage in "<activity...>" of "tk.tcl.wish.AndroWish" in "AndroidManifest.xml" should be replaced by "de.elmicron.hibiscan.HibiScan". Here, this is done in the next step.

Remark: the usage of a derived class did not make any difference to me. I could stay with the class "tk.tcl.wish.AndroWish". Nevertheless, Christian: recommends it. Comments welcome...

Start script directly

Loose translation of E-Mail from Christian::

The file "AndroidManifest.xml" for own applications should bette be structured similar to ".../hellotcltk/AndroidManifest.xml". The own application should not be started by the activity "AndroWishLauncher", but better directly, using the remaining intent filter:

```
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
```

So, within the "AndroidManifest.xml" file, there are the following changes:

- Use only one activity with the new class and the proposed intent-filter.
- I changed the product version and class to 6.0 and numeric 600, as this is the port of an existing program, which has version number 6. and the following changes already in other sections:
- Use package name "de.elmicron.hibiscan"
- Use class "de.elmicron.hibiscan.HIBIScan" instead "tk.tcl.wish.AndroWish"
- Only minimal permissions

The resulting file looks like that:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="de.elmicron.hibiscan"
          android:installLocation="auto"
```

```
          android:versionCode="600"
          android:versionName="6.0">
  <application android:label="@string/app_name"
               android:icon="@drawable/androwish"
               android:allowBackup="true"
               android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
               android:hardwareAccelerated="true">
    <activity android:name="de.elmicron.hibiscan.HIBIScan"

android:configChanges="orientation|keyboardHidden|keyboard|screenSize|mnc|mcc|locale|fontScale|uiMode"
              android:label="@string/app_name">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

  <!-- Android 2.3.3 -->
  <uses-sdk android:minSdkVersion="9" android:targetSdkVersion="14" />

  <!-- OpenGL ES 2.0 -->
  <uses-feature android:glEsVersion="0x00020000" />

  <!-- USB support -->
  <uses-feature android:name="android.hardware.usb.host" />

  <!-- Disable screen compatibility modes -->
  <supports-screens android:smallScreens="true"
                    android:normalScreens="true"
                    android:largeScreens="true"
                    android:xlargeScreens="true" />

  <!-- Allow writing to external storage etc. -->
  <uses-permission android:name="android.permission.INTERNET" />
</manifest>
```

Resources

Change the AppName in res/values/strings.xml

Change the Androwish icons in res/drawable-*/androwish.png (Resolutions: 72x72, 48x48, 96x96, 144x144).

Remove fonts

Christian: suggestion via E-Mail: 2 additional MB's may be economized by not including the font folder ".../jni/sdl2tk/library/fonts" as follows:

```
cd jni/sdl2tk
mv library/fonts .
```

In this case, the buildin Droid* fonts are used as fallback which are included in Android firmware. They don't look so much less attractive...

This results in a package file size of 7.8MB

On Android 5, this requires Don Quixote release (2015-04) of Androwish to run. Otherwise, Androwish does not start on Android 5.

Release signing

Create a release key by (replace "elmicron" by your own name):

```
cd ~/android
keytool -genkey -v -keystore android_elmicron.keystore -alias android_elmicron -keyalg RSA -keysize
2048 -validity 10000
```

You get promted to a keystore password and the key values. I only filled common name and Organisation. Then you get prompted to a key password.

This generates the file "~/android/android_elmicron.keystore".

Then add those lines to "~/android/androwish/ant.properties":

```
key.store=../android_elmicron.keystore
key.alias=android_elmicron
key.store.password=<mypw1>
key.alias.password=<mypw2>
```

and do

```
ant release
```

The final apk is in "bin/AndroWish-release.apk".

2015-06-04 Harald Oehlmann

Building AndroWish

Building AndroWish

Requirements

- Android SDK (version 12 or later)
- Android NDK (r7 or later)
- Minimum API level support by SDL is 10 (Android 2.3.3), requested API level from `project.properties` is 16 (Android 4.1)
- CPUs supported for native shared libraries are currently `armeabi` and `x86`. This can be changed in `jni/Application.mk`.

Building and Running AndroWish

Old school using Apache `ant`:

1. Refresh the project settings using the android command from Android SDK: `android update project`
2. Review `local.properties` to point to the directory where the Android SDK resides.
3. Use `ant` to build AndroWish from scratch: `ant debug`. This includes building the C libraries using Android NDK. That step can be performed separately by running `ndk-build` in the `jni` directory or by invoking `ant ndk-build`
4. The resulting Android APK is built to `bin/AndroWish-debug.apk` which can be installed onto a device or emulator using `adb install -r bin/AndroWish-debug.apk`.
5. Start AndroWish on device or emulator using `adb` from the development system: `adb shell am start tk.tcl.wish/.AndroWishLauncher`.
6. Clean the build tree with `ant clean`.

New style using `gradlew`:

1. Setup your environment regarding `ANDROID_HOME` and the `ndk-build` command e.g. by setting both a proper `PATH` and `ANDROID_NDK_HOME`.
2. Use `gradlew` to build AndroWish from scratch: `./gradlew assembleDebug`. As above this performs both the NDK build and the final compile and packaging steps.
3. The resulting Android APK is built to `build/outputs/apk/AndroWish-debug.apk` which can be installed onto a device or emulator using `adb install -r build/outputs/apk/AndroWish-debug.apk`.
4. Start AndroWish on device or emulator using `adb` from the development system: `adb shell am start tk.tcl.wish/.AndroWishLauncher`.
5. Clean the build tree with `./gradlew clean`.

Building vanillawish/undroidwish on Windows

By Stephan Effelsberg

The crucial part in building undroidwish on Windows is the setup of the environment. This set of instructions is the result from building undroidwish on Windows 7 Pro 32 using MSys2.

Tool: MSys2

Install MSys2 from the homepage or use a package manager like Chocolatey. Once you have MSys2 you can install any necessary package via its package manager pacman, e.g

```
    pacman -S <name of package>
```

Some tools like **CMake** are nice to have a system-wide install. In this case just make sure that the tools are listed in the PATH.

Tool: MinGW

Install MinGW, then copy some of the binaries to give them their necessary names, see for example these instructions on the Enlightenment wiki. Unfortunately, the binaries of a Windows installation of MinGW don't have the names of the cross compiler suite of MinGW. Consult the build script that you're finally going to call to learn about the names of the individual binaries. This is an excerpt from build-undroidwish-win32.sh:

```
# the toolchain
if test -d /opt/mingw64/bin ; then
  # use -march=i386 -mtune=i386 for Win2000 and/or old CPUs w/o
  # SSE like VIA C3
  echo using toolchain from /opt/mingw64/bin
  PATH="/opt/mingw64/bin:$PATH"
  STRIP="x86_64-w64-mingw32-strip"
  OBJCOPY="x86_64-w64-mingw32-objcopy"
  AR="x86_64-w64-mingw32-ar"
  RANLIB="x86_64-w64-mingw32-ranlib"
  CC="x86_64-w64-mingw32-gcc -m32 -march=i386 -mtune=i386 -DTCL_UTF_MAX=3"
  CC_OLD="x86_64-w64-mingw32-gcc -m32 -march=i386 -mtune=i386 -D_WIN32_WINNT=0x0400 -DTCL_UTF_MAX=3"
  CXX="x86_64-w64-mingw32-g++ -m32 -march=i386 -mtune=i386 -fno-exceptions -DTCL_UTF_MAX=3"
  RC="x86_64-w64-mingw32-windres -F pe-i386"
  NM="x86_64-w64-mingw32-nm"
  export STRIP OBJCOPY AR RANLIB CC CC_OLD CXX RC NM
else
  # would like to use -march=i386 -mtune=i386, too, but then gcc-4.8
  # cannot link due to missing atomic support for this CPU, thus must
  # have Pentium at least
  echo using toolchain prefix i686-w64-mingw32
  STRIP="i686-w64-mingw32-strip"
  OBJCOPY="i686-w64-mingw32-objcopy"
  AR="i686-w64-mingw32-ar"
  RANLIB="i686-w64-mingw32-ranlib"
  CC="i686-w64-mingw32-gcc -m32 -march=i586 -mtune=generic -DTCL_UTF_MAX=3"
  CC_OLD="i686-w64-mingw32-gcc -m32 -march=i586 -mtune=generic -D_WIN32_WINNT=0x0400 -DTCL_UTF_MAX=3"
  CXX="i686-w64-mingw32-g++ -m32 -march=i586 -mtune=generic -fno-exceptions -DTCL_UTF_MAX=3"
  RC="i686-w64-mingw32-windres -F pe-i386"
  NM="i686-w64-mingw32-nm"
  TWAPI_LDFLAGS="-L${AWDIR}/undroid/compat/win32/lib32"
  export STRIP OBJCOPY AR RANLIB CC CC_OLD CXX RC NM TWAPI_LDFLAGS
fi
```

What if ... I instead rename the environment variables in the script to reflect the names of the binaries?

When compiling libwebsockets, you may encounter a strange case of `CMAKE_AR-NOTFOUND`. I don't know why `ar` is so special to CMake but if you search for it you can find many surprised developers who stumbled upon it.

Tool: CMake

Get it from cmake.org or a package manager.

Tool: rsync

For calling `init` of the build script.

```
    pacman -S rsync
```

Tool: make

```
    pacman -S make
```

Tool: nasm

nasm.us or via package manager. Needed for jpeg-turbo.

```
    pacman -S nasm
```

Tool: Perl

```
    pacman -S perl
```

Tool: bc

`curl` calls `curl-config` and this script needs `bc` to calculate the version number requirements.

```
    pacman -S bc
```

Tool: pkg-config

```
    pacman -S pkg-config
```

What if ... I forget pgk-config?

You will not get error messages but some modules will silently be ignored, e.g. the jsmpeg video driver. You will only learn about this when trying to use an ignored module.

Tool: texinfo

`ffidl` may complain about a missing `makeinfo`.

```
    pacman -S texinfo
```

Starting the shell

There are some options to start the MSys shell (and you surely have already started one in order to install the tools). Make sure that the shell is being run as an MSys shell, not a MinGW shell, by checking `uname`. The result should look like

```
MSYS_NT_6.1-7601
```

and not like

```
MINGW_NT_6.1-7601
```

Now follow the simple build instructions to get your wish.

What if ... I run a shell in MinGW mode?

Some modules may give you error messages like "Please use win32/Makefile.gcc instead." or "... is not a cygwin compiler."

dmtx command

dmtx command

Name

dmtx::* - interface to the libdmtx.org Data Matrix Code scanner library.

Synopsis

```
package require dmtx
dmtx::decode ?options?
dmtx::async_decode ?options?
```

Description

These commands are used to scan Data Matrix Codes off pixel image data.

`dmtx::decode` *`photoEtc ?scale timeout?`*

> Scans the photo image *`photoEtc`* for Data Matrix Code information. Alternatively, *`photoEtc`* can be a four element list describing a greyscale or RGB image as a byte array. The elements must be width, height, depth and byte array of the image in this order. The optional integer parameter *`scale`* downsamples the image before the scan takes place. The optional *`timeout`* limits the scan process to that many milliseconds. The command returns a three element list made up of a flag indicating success (=1) or failure (=0) of the scan process, the amount of milliseconds spent on decoding, and the scan result as a byte array.

`dmtx::async_decode` *`photoEtc callback ?scale timeout?`*

> Similar to `dmtx::decode` but the decoder is run as a background thread and the result is presented to a *`callback`* procedure. It requires the Tcl core being built with thread support, and a running event loop since the callback is invoked as an event or do-when-idle handler. Three additional arguments are passed to *`callback`*: a flag indicating success (=1) or failure (=0) of the scan process, the number of milliseconds for decoding, and the scan result as a byte array. The optional parameters *`scale`* and *`timeout`* have the same meaning as in the `dmtx::decode` command. The default timeout value is 1000 milliseconds. Caution: only a single thread instance is supported per Tcl interpreter, i.e. another asynchronous decode process can only be started when a previous decode process has finished.

`dmtx::async_decode abort`

> Aborts a running asynchronous decode process.

`dmtx::async_decode status`

> Returns the current state of the asynchronous decode as a string: *`stopped`* when no asynchronous decode thread has been started, *`running`* when a asynchronous decode is in progress, and *`ready`* when the next asynchronous decode can be started.

`dmtx::async_decode stop`

> Stops the background thread for asynchronous decoding if it has been implicitely started by a prior `dmtx::async_decode`. This can be useful to conserve memory resources.

Environment Variables

Environment Variables

Some environment variables in the `env` array are setup on early startup of AndroWish.

`env(EXTERNAL_FILES)`

App specific directory on external storage.

`env(EXTERNAL_STORAGE)`

Path name of external storage (could be internal SD card).

`env(EXTERNAL_STORAGE2)`

Path name of external storage (real external SD card).

`env(HOME)`

App's home directory (internal storage), usually `/data/data/tk.tcl.wish/files`.

`env(INTERNAL_STORAGE)`

App specific directory on internal storage (identical with `$env(HOME)`).

`env(LANG)`

System language.

`env(LD_LIBRARY_PATH)`

Load path for shared libraries including app specific directory (usually `/data/data/tk.tcl.wish/libs`).

`env(OBB_DIR)`

On some Android versions extra stuff bundled with the app (currently unused).

`env(PACKAGE_CODE_PATH)`

Path name of the app's APK.

`env(PACKAGE_NAME)`

Package name where the app's main class comes from (`tk.tcl.wish`).

`env(PATH)`

Path for `exec(n)` including app specific directory

`env(TMPDIR)`

Path name for temporary files (usually `/data/data/tk.tcl.wish/cache`, fallback is value of `$env(HOME)`).

To test if a Tcl script is executing on the Android platform `sdltk android` (see sdltk command) should be used.

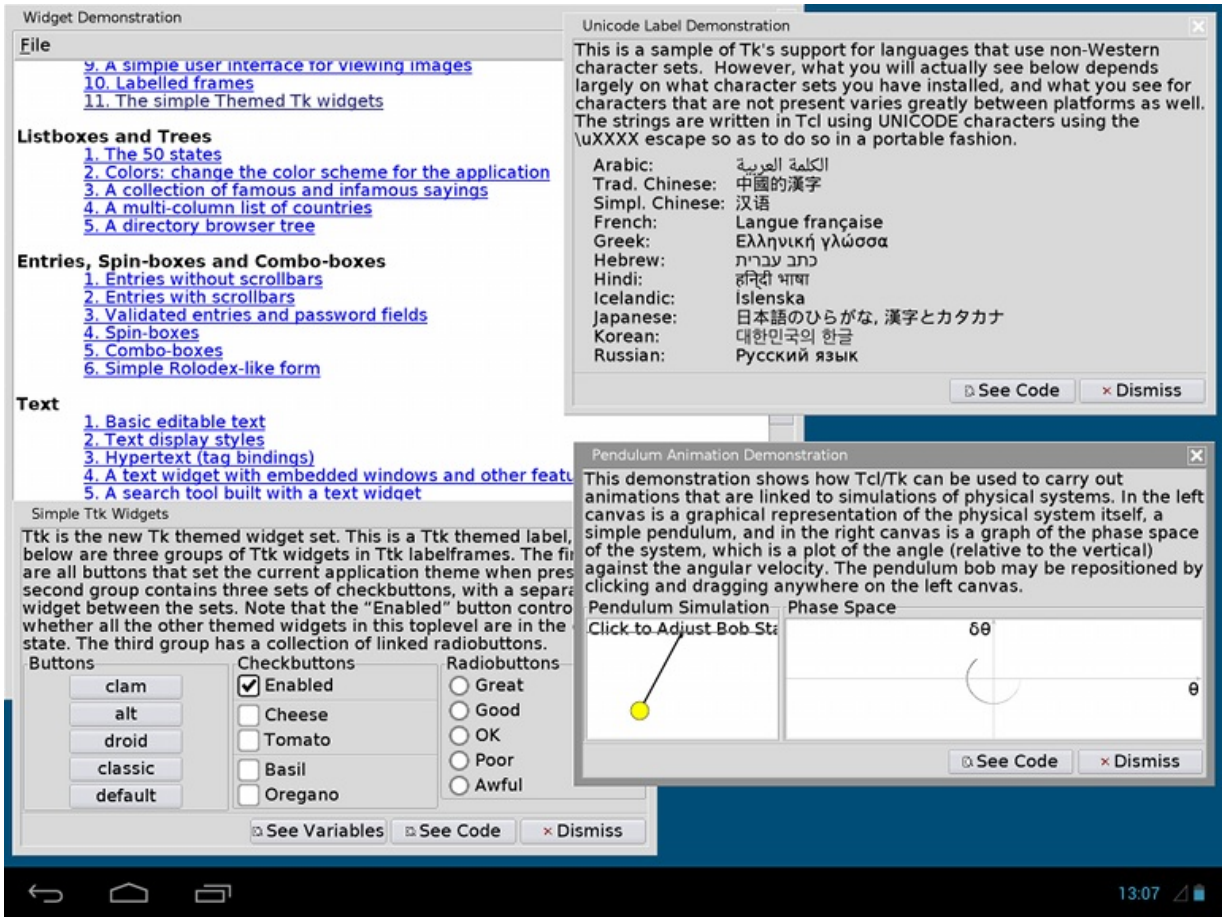
Example Scripts

Example Scripts And Screenshots

Many packaged example scripts can be invoked on typical Android devices by a "`androwish:///<pathname>`" URL on the VFS mounted `/assets` folder. This works with the Android web view component, Firefox, and Chrome. For Firefox, the last path component (the tcl file) must be URL encoded, e.g. `test.tcl` must be written as `test%2Etcl`.

Tk widget demo

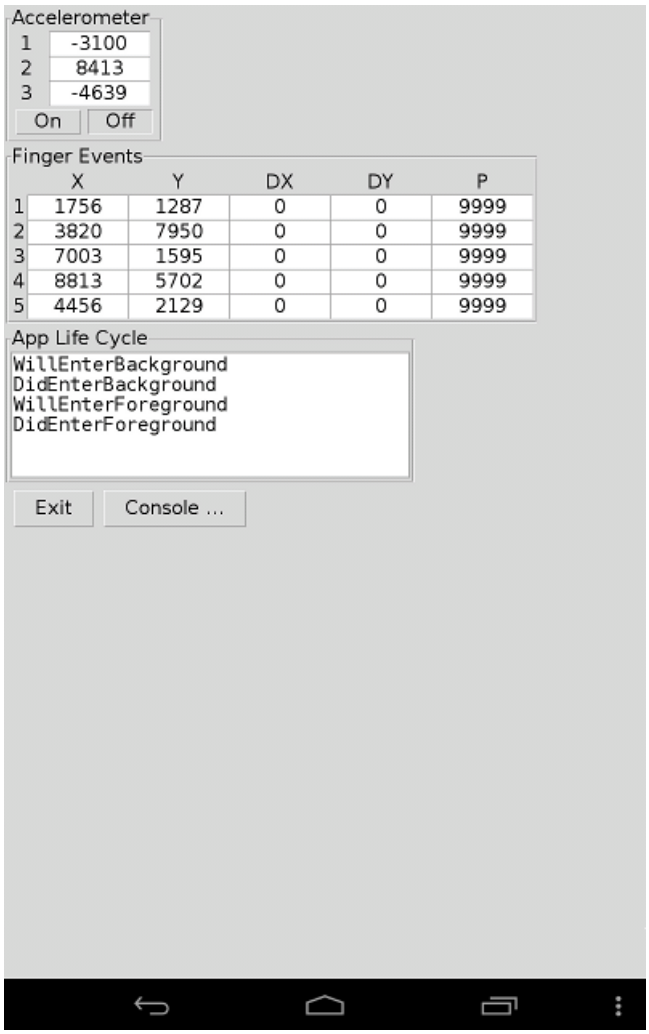
> **androwish:///assets/sdl2tk8.6/demos/widget**, **source code**



Screenshot taken on an i-onik TP9.7-1200QC-Ultra tablet

App Life-cylce, accelerometer, finger events

> **androwish:///assets/sdl2tk8.6/demos/android_demo%2Etcl**, **source code**



Screenshot taken on a Lenovo Yoga 8 tablet

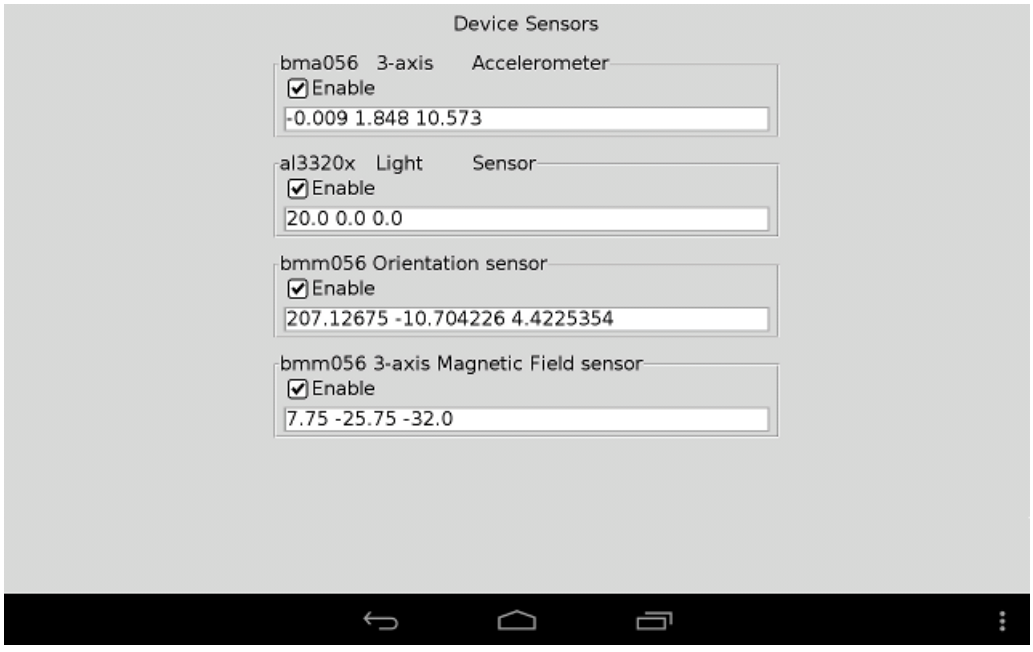
Accelerometer with canvas widget

[androwish:///assets/sdl2tk8.6/demos/android_accel%2Etcl](), [source code]()

Screenshot taken on a Lenovo Yoga 8 tablet

Device sensors

androwish:///assets/sdl2tk8.6/demos/android_sensors%2Etcl, source code



Screenshot taken on a Lenovo Yoga 8 tablet

Compass using magnetic field sensor and accelerometer

androwish:///assets/sdl2tk8.6/demos/android_compass%2Etcl, source code

Screenshot taken on a Lenovo Yoga 8 tablet

Pinch-to-zoom with canvas widget

androwish:///assets/sdl2tk8.6/demos/android_zoom%2Etcl, source code

Eliza: speech recognition and speech-to-text

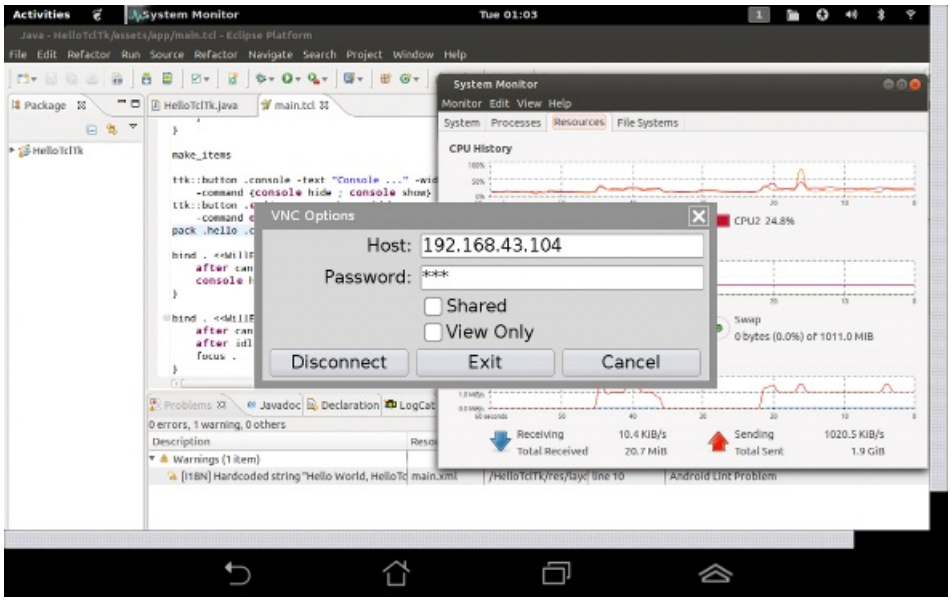
androwish:///assets/sdl2tk8.6/demos/android_eliza%2Etcl, source code

TclMixer: audio output and mod music playback

androwish:///assets/tclmixer1.2.3/test%2Etcl, source code

Simple VNC viewer

androwish:///assets/vnc0.4/vncviewer%2Etcl, source code



Screenshot taken on ASUS Fonepad K004 ME371G

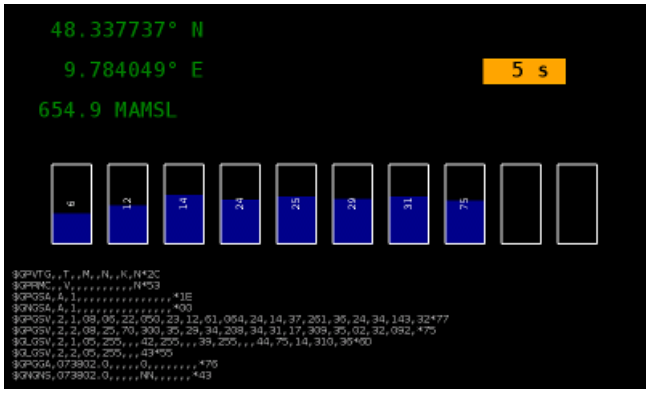
Barcode generation using ZINT

androwish:///assets/zint2.5.0/demo%2Etcl, source code

Screenshot taken on a HTC One V smartphone

GPS/NMEA display

androwish:///assets/sdl2tk8.6/demos/android_gps%2Etcl, source code



Screenshot taken on a HTC One V smartphone

Walkie-talkie using snack and UDP multicast

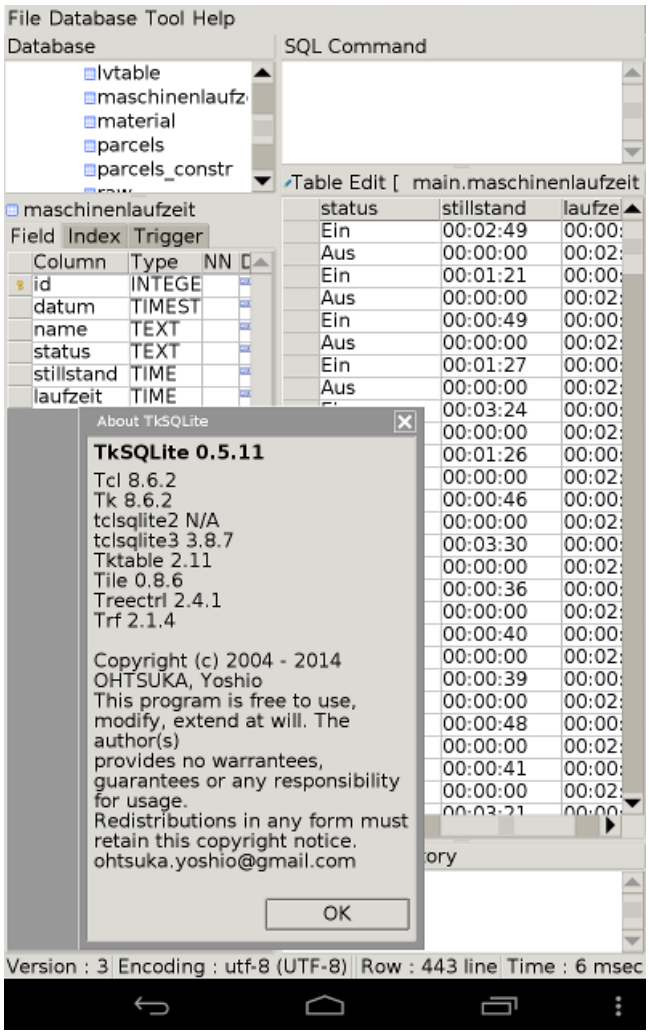
androwish:///assets/snack2.2.10/tcl_talkie%2Etcl, source code

Simple chat using Bluetooth serial port profile (SPP)

androwish:///assets/sdl2tk8.6/demos/android_btchat%2Etcl, source code

TkSQLite database frontend for SQLite

androwish:///assets/tksqlite0.5.11/tksqlite%2Etcl, source code



Screenshot taken on a Lenovo Yoga 8 tablet

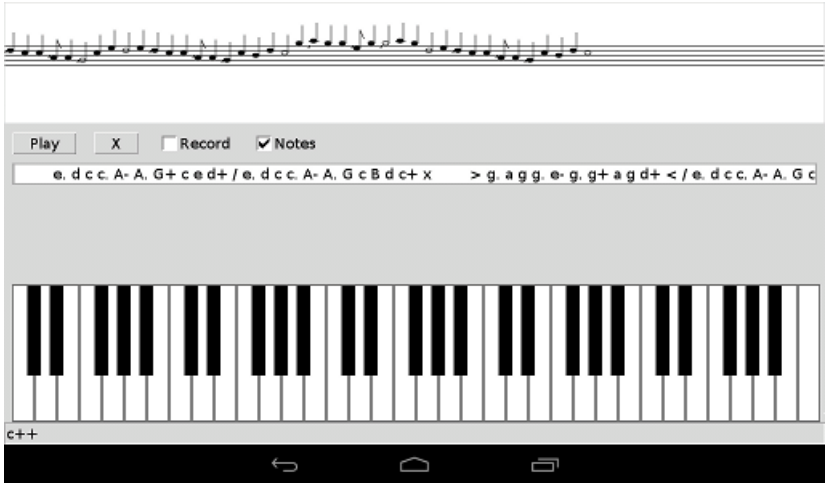
Canvas 3D using camera for texturing cube surfaces

[androwish:///assets/Canvas3d1.2.1/demo/photocube%2Etcl](), [source code]()

Screenshot taken on a Lenovo Ideatab A3000-H tablet

Interactive MIDI music

[androwish:///assets/music0.1/music%2Etcl](), [source code]()



Screenshot taken on a Lenovo Ideatab A3000-H tablet

Piano, a pocket synthesizer

androwish:///assets/music0.1/piano%2Etcl, source code

Data Matrix Code scanner using camera and dmtx command

androwish:///assets/dmtx0.7.5/android_demo%2Etcl, source code

Barcode scanner using camera and zbar command

androwish:///assets/zbar0.10/android_demo%2Etcl, source code

Minimalist WebCam in ≈ 100 LOC

androwish:///assets/sdl2tk8.6/demos/android_webcam%2Etcl, source code

Tkbugz, a game requiring a VR headset and a USB or Bluetooth joystick

androwish:///assets/tkbugz/vr_bugz%2Etcl, source code

jsmpeg SDL Video Driver

jsmpeg SDL Video Driver

An experimental SDL video driver named **jsmpeg** is provided since Valentine's Day 2019. It uses the technique described in https://github.com/phoboslab/jsmpeg and https://github.com/phoboslab/jsmpeg-vnc in combination with HTML5 and WebGL in a modern browser to provide display, mouse, and keyboard to a normal undroidwish.

This means, that the rendering is performed into a memory buffer, which is encoded into a modified MPEG-1 transport stream, sent over a Websocket to a web browser, which performs MPEG-1 decoding and rendering into a HTML5 canvas optionally using WebGL. Likewise, mouse and keyboard events are sent on the same Websocket from the browser back to the **jsmpeg** driver, transformed to SDL mouse and keyboard events and further processed by the undroidwish application.

Frame rate and required bandwidth are moderate. Currently, 25 frames per seconds are sent at most, which require some few hundred kilobits per second. Since April 2019 limited support for OpenGL is available for the Canvas3D and tkZinc widgets. It requires a working EGL/OpenGL infrastructure (Linux etc.) or Windows OpenGL. On Linux this normally requires an X11 display connection except for very recent versions of Mesa and GPU drivers, i.e. on Debian 10 and Fedora 30 the environment variable **EGL_DISPLAY** can be set to **surfaceless** in order to turn on headless GPU mode.

For the adventurous, there is a test version for Linux x86_64 (Debian 9, Fedora 30, CentOS 7), Windows 32 bit (XP or newer), Windows 64 bit (XP or newer), and MacOSX (tested on High Sierra). All can be run using the **jsmpeg** video driver when the environment variable **SDL_VIDEODRIVER** has the value **jsmpeg** and the required FFMpeg DLLs/shared libraries are available on the system, e.g.

```
    # POSIX
    export SDL_VIDEODRIVER=jsmpeg
    ./undroidwish-x86_64-deb9 builtin:widget -sdlwidth 800 -sdlheight 600
```

```
    REM Windows
    SET SDL_VIDEODRIVER=jsmpeg
    undroidwish-win32.exe builtin:widget -sdlwidth 800 -sdlheight 600
```

```
    # Mac OSX
    export SDL_VIDEODRIVER=jsmpeg
    /Applications/undroidwish.app/Contents/MacOS/undroidwish builtin:widget -sdlwidth 800 -sdlheight
600
```

For Windows, the required DLLs are **avutil-56.dll**, **avcodec-58.dll**, **swresample-3.dll**, and **swscale-5.dll** which are available from https://www.ffmpeg.org/download.html and preferably loaded from **%PROGRAMFILES%\ffmpeg\bin**. For Linux, the shared libraries are available per installing the distribution's ffmpeg package(s). For MacOSX, the homebrew ffmpeg package provides the necessary shared libraries.

By default, the HTTP/Websockets port is 8080 which can be overridden with the environment variable **SDL_VIDEO_JSMPEG_PORT**. Thus, the URL

```
    http://localhost:8080
```

connects the browser with the **jsmpeg** enabled undroidwish. If the browser's WebGL implementation isn't suitable for proper displaying the undroidwish root window, the alternate URL

```
    http://localhost:8080/?use2d
```

turns off WebGL usage.

Note that all local TCP/IP addresses are bound, not just localhost. Thus, if your local TCP/IP address is 192.168.1.9, then http://192.168.1.9:8080 will work, and provides access to your app via non-localhost devices.

HTTP AUTH

You can optionally password-protect the http port serving your app, by defining the **SDL_VIDEO_JSMPEG_AUTH** variable like so:

```
export SDL_VIDEO_JSMPEG_AUTH=$(echo -n user:pass | base64)
```

This will cause your web browser to require a username/password combination in order to display the page. For more info about HTTP AUTH see the Wikipedia article Basic_access_authentication.

Screen recording

You can optionally save your entire Tk app's running to a MPEG1 file, by defining the **SDL_VIDEO_JSMPEG_OUTFILE** like so:

```
export SDL_VIDEO_JSMPEG_OUTFILE=data.mpg
```

HTML Page Title

The HTML generated to support the jsmpeg driver will automatically use the Tk window manager title to define your web page's web title. Thus, this command in your Tcl/Tk program:

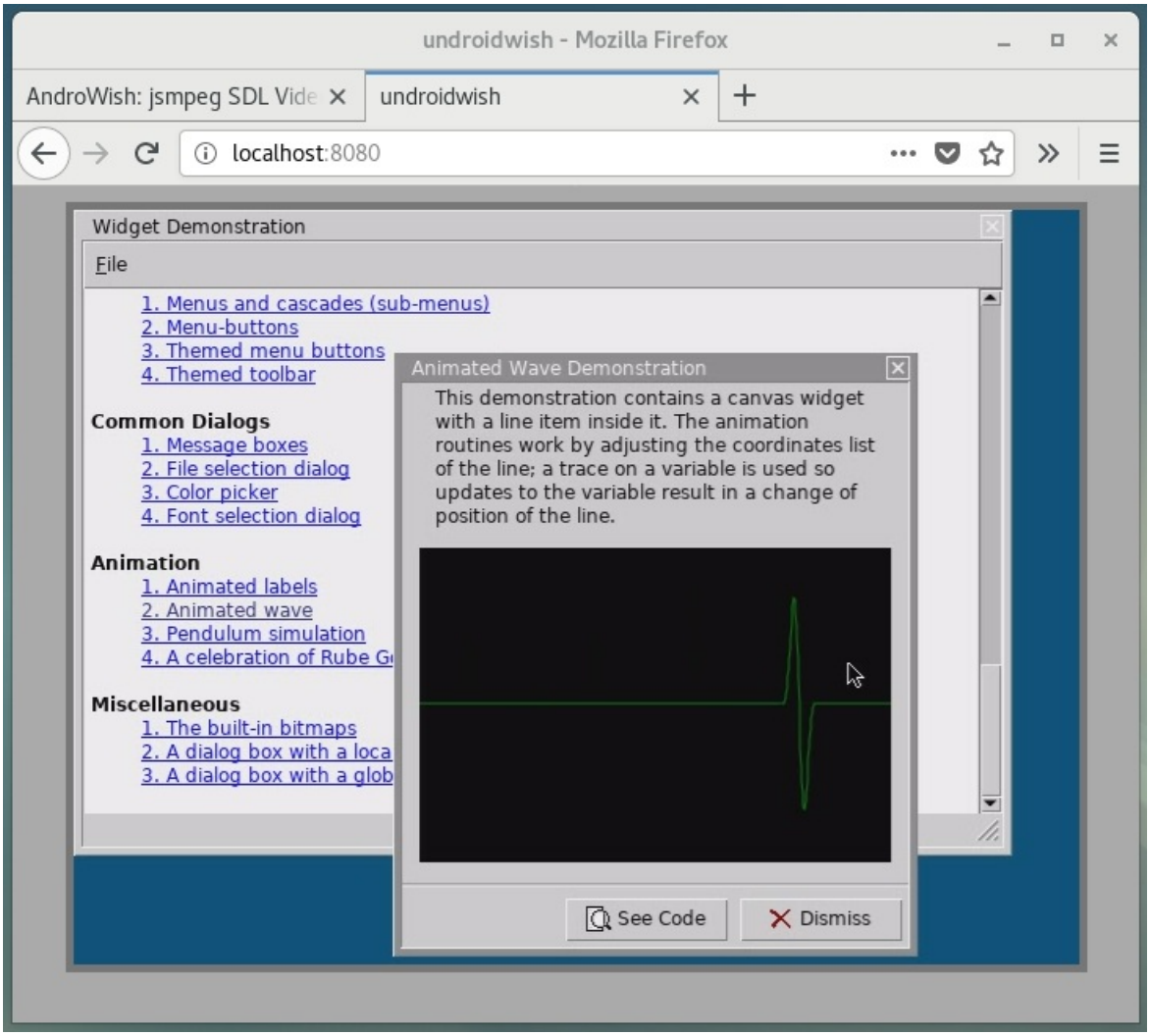
```
wm title . "Hello World \U1F601"
```

will result in your web browser titling this page tab as "Hello World 😁". Only the application's toplevel window (the "." in Tk parlance) will set the browser title.

A note about colors

Due to RGB->YCbCr->RGB color conversions taking place when using the **jsmpeg** driver, the output colors of your app (when viewed in a web browser) are never exactly what they would be in Tk.

For example, white #FFFFFF will render as #FEFEFE. This difference is very slight.



Screenshot taken in a GNOME Wayland session

Limitations of AndroWish

Limitations of AndroWish

- ~~The X11 emulation is not thread-safe, thus~~ it is ~~im~~possible to do a `package require Tk` from another thread. ~~But~~ multiple Tcl `interps` in the main thread work. Since "The Flintstones (2014-09-30)" release the X11 emulation is thread safe but many extensions are not, e.g. snack, expect etc. Your mileage may vary.
- Due to Android process start up with respect to the window system the Tcl `exec` command cannot be used to start other Tk processes.
- The bandwidth of device screen resolutions is broad (100 dpi .. 500 dpi) compared to usual desktop systems. But many elements of Tk widgets are pixel based. This is partly addressed by using icon bitmaps in various sizes but far from being a perfect solution.

Make minimal vanillawish binary

How to make a minimal vanillawish binary ...

... out of a running vanillawish? The same technique can be used with undroidwish, too.

```
# we're working in a temp directory ...
file delete -force -- tmpdir[pid]

# extract everything from current binary
file copy [info nameofexecutable] tmpdir[pid]

# remove unwanted stuff from tmpdir[pid]
# ... TODO (adapt it to your requirements)
# here let's keep the bare minimum
apply {keep {
    foreach name [glob -tails -dir tmpdir[pid] *] {
        if {$name in $keep} {
            continue
        }
        file delete -force -- [file join tmpdir[pid] $name]
    }
}} [list tcl8.6 tk8.6 sdl2tk8.6]

# the app directory to place our own stuff
file mkdir [file join tmpdir[pid] app]

# add app code to tmpdir[pid]/app, at least a main.tcl
# is required which provides the entry point
# ... TODO (example given)
apply {name {
    set f [open $name w]
    # a very small app ...
    puts $f "button .b -text {Press Me} -command exit ; pack .b"
    close $f
}} [file join tmpdir[pid] app main.tcl]

# rebuild a new binary, stripping the temp directory in all file names
if {$tcl_platform(platform) eq "windows"} {
    zipfs::mkimg myapp.exe tmpdir[pid] tmpdir[pid]
} else {
    zipfs::mkimg myapp tmpdir[pid] tmpdir[pid]
}

# remove the temp directory
file delete -force -- tmpdir[pid]

# try to start the new binary if it is a wish
# otherwise stdin would be needed and we can't
# run in background.
if {[info command winfo] eq "winfo"} {
    if {$tcl_platform(platform) eq "windows"} {
        catch {exec [file join [pwd] myapp.exe] < NUL: &}
    } else {
        catch {exec [file join [pwd] myapp] < /dev/null &}
    }
}

# done
exit
```

modbus

modbus command

Name

modbus - Tcl interface to libmodbus

Synopsis

```
package require Tcl 8.6
package require modbus
modbus::new cmd host service
modbus::new cmd serial baud parity data stop ?slave_addr?
cmd destroy
cmd connect
cmd close
cmd setchan chan
cmd response_timeout ?ms?
cmd serial_mode ?mode?
cmd read_bits addr ?number?
cmd read_input_bits addr ?number?
cmd read_registers addr ?number?
cmd read_input_registers addr ?number?
cmd write_bit addr value
cmd write_register addr value
cmd write_bits addr value ...
cmd write_registers addr value ...
cmd set_slave slave_addr
```

Description

This package provides a Tcl interface to libmodbus (see http://libmodus.org) using Ffidl and TclOO.

Commands

`modbus::new` *`cmd host service`*

> Creates a new command *`cmd`* which implements a Modbus-TCP connection object to the given *`host`* (IP address or hostname) and *`service`* (symbolic or numeric TCP port). Further operations on that object are carried out by invoking methods on *`cmd`*.

`modbus::new` *`cmd serial baud parity data stop ?slave_addr?`*

> Creates a new command *`cmd`* which implements a Modbus-RTU connection object on the serial line *`serial`* with parameters *`baud`* rate, *`parity`* (N=none, O=odd, E=even), *`data`* bits, and *`stop`* bits. The optional parameter *`slave_addr`* specifies the Modbus-RTU slave address and defaults to zero. Further operations on that object are carried out by invoking methods on *`cmd`*.

`cmd` `destroy`

> Destroys the connection object *`cmd`*, releases resources and closes communications links.

`cmd` `connect`

> Connects the connection object *`cmd`* to its peer (a TCP server for Modbus-TCP or a serial line for Modbus-RTU).

`cmd` `close`

> Closes the connection (either the socket or the serial line) of the connection object *`cmd`*.

`cmd` `setchan` *`chan`*

> On POSIX platforms, this method duplicates the operating system handle of the Tcl channel *`chan`* and wraps it into the *`cmd`* connection object. The Tcl channel can be closed immediately after this operation. Depending on the constructor, the operating system handle must provide socket or tty semantics for further I/O methods on *`cmd`* to succeed. On Windows platforms, this method is not supported.

`cmd` `response_time` *`?ms?`*

> Queries or sets the response timeout on the connection object *`cmd`*. The timeout is specified in milliseconds.

`cmd` `serial_mode` *`?mode?`*

Queries or sets RS-232 or RS-485 mode on the Modbus-RTU connection object *cmd*. For RS-232 mode must be 0, for RS-485 it must be 1.

cmd read_bits *addr ?number?*

Reads *number* coil status bits starting with address *addr* from the connection object *cmd*. Number defaults to one.

cmd read_input_bits *addr ?number?*

Reads *number* input status bits starting with address *addr* from the connection object *cmd*. Number defaults to one.

cmd read_registers *addr ?number?*

Reads *number* holding registers starting with address *addr* from the connection object *cmd*. Number defaults to one.

cmd read_input_registers *addr ?number?*

Reads *number* input registers starting with address *addr* from the connection object *cmd*. Number defaults to one.

cmd write_bit *addr value*

Writes *value* into the coil status bit with address *addr* on the connection object *cmd*.

cmd write_register *addr value*

Writes *value* into the holding register with address *addr* on the connection object *cmd*.

cmd write_bits *addr value ...*

Writes one ore more values into the coil status bits starting with address *addr* on the connection object *cmd*.

cmd write_registers *addr value ...*

Writes one ore more values into the holding registers starting with address *addr* on the connection object *cmd*.

cmd set_slave *?slave_addr?*

Sets the slave address for Modbus-RTU on the connection object *cmd* to *slave_addr*.

Muzic MIDI sound package

muzic command

Name

muzic - a MIDI sound package compatible with Muzic.

Synopsis

```
package require Muzic
muzic::subcommand ...
```

Description

muzic is a Tcl music interface to the Sonivox MIDI rendering (software synthesis) library on Android. The original package was developed by Steve Landers and is Copyright (c) 2005 Eolas Technologies Inc. It is released under a Tcl/BSD style license.

To use Muzic, no special provisions are needed, since it is fully integrated in AndroWish.

The Muzic API contains just five procedures:

`muzic::init`

> Must be called once to initialize audio playback.

`muzic::soundfont file`

> This command exists for compatibility with the original package. It can be called with no arguments or with `builtin`. Everything else throws an error since the Android software synthesizer has no support for SoundFont files.

`muzic::channel` *`channel instrument`*

> Assigns an instrument to a channel. *`channel`* is an integer from 0 to 15 identifying the MIDI channel. *`instrument`* is the instrument number, typically a MIDI instrument number from 0 to 127.

`muzic::playnote` *`channel pitch volume ?duration?`*

> Plays a note on specified *`channel`*, at specified *`pitch`* and *`volume`*. The pitch is the raw MIDI pitch, as per the general midi standard - where middle C is 60 (see http://www.mozart.co.uk/information/articles/midinote.htm which has a table of MIDI pitch values). *`volume`* is a number between 0 and 100. *`duration`* is optional, and defaults to 500 (i.e. 500 ms). If a negative duration is given, the note is played continuously. If a volume of zero is given, playback of the note ends.

`muzic::close`

> To be called when MIDI audio playback shall be stopped in order to conserve battery power.

Releases

List of AndroWish Releases

Fossil Tag	Date	Remarks
The Flux Capacitor	2020-11-05	The first release of AndroWish in 2020 featuring Tcl/Tk 8.6.10, SQLite 3.33.0, SDL 2.0.6 with patches, and many other updated packages. Some final and draft TIPs have been backported or added (160, 262, 302, 431, 456, 462, 496, 511, 517, 565, 574, and 586).
Eppur si muove	2019-06-22	The first release of AndroWish in 2019 featuring Tcl/Tk 8.6.9, SQLite 3.28.0, SDL 2.0.6 with patches, and many other updated packages. Some new extensions are included: tkvlc, topcua, tclJBlend, and tcl-fuse. A webview for the major desktop platforms is contained as a preview. A new SDL video driver called "jsmpeg" is included in the undroidwish builds.
Asteroid Day	2018-06-30	Another low-impact release of AndroWish featuring Tcl/Tk 8.6.8, SQLite 3.24.0, SDL 2.0.6 with patches, and many other updated packages. Some new extensions are included: tclcan, modbus, snap7, and fswatch. Build support for multiple platforms is more stable. The Wayland variant now runs on CentOS 7.5, too, plus a frame buffer rendering mode based on Linux kernel mode setting with direct render mode is included.
The Leyden Jar	2017-10-11	The long overdue release of AndroWish in 2017 featuring Tcl/Tk 8.6.7, SQLite 3.20.1, SDL 2.0.5, Tkzinc 3.3.6 and many other updated packages. Other highlights are: basic NFC support in AndroWish, stereoscopic render modes both in the SDL based Tk and in the 3D canvas widget, build support for more platforms in undroidwish, e.g. FreeBSD, OpenIndiana, MacOSX, Haiku, plus support for Wayland.
Bonfire Night	2016-11-05	This is the 3rd anniversary edition of AndroWish featuring Tcl/Tk 8.6.6, SQLite 3.15.1, and tksvg. Many other packages are updated to newer versions, too. The undroidwish based builds now contain the same subset of BLT (barchart, graph widgets) as AndroWish. All builds contain a proposed TIP#302 implementation to be indifferent with respect to wall clock changes by making computation of durations based on a monotonic clock source if supported by the OS.
The Wow! Signal	2016-08-15	This is an update release featuring Tcl/Tk 8.6.6, SQLite 3.14.1, and LibreSSL 2.2.9. Many other packages are updated to newer versions, too. The undroidwish based builds now contain TWAPI/WiTS (Windows versions) and broader support for video capture (both, Windows and Linux).
El Caballero de la Triste Figura	2016-04-23	This an update release featuring Tcl/Tk 8.6.5 and SQLite 3.12.2. Some other packages are updated to newer versions, too (SDL 2.0.4 plus patches, tcllib, pdf4tcl). OpenSSL is replaced by LibreSSL 2.2.6 and TkHTML 3 is added. This release introduces undroidwish plus an AndroWish SDK based on undroidwish binaries.
Caractacus Potts	2015-12-18	This is mainly a bug fix and update release. Some packages are updated to newer versions (SQLite 3.9.2, gridplus 2.11, icons 2.0, libpng 1.2.54), the "ble" command and underlying infrastructure is more stable, the "borg" command is improved and allows now to turn Bluetooth on and off and to send SMS. The tkpath widget combined with pdf4tcl now can generate PDF documents from most supported item types including alpha blending and color gradients. The AndroWish SDK is improved and now able to run on all supported Tcl/Tk desktop platforms.

Back to the Future	2015-10-21	This release adds full Unicode 8.0 support including Emojis by using 32 bits for the internal representation of Unicode codepoints and up to 4 byte long UTF-8 sequences (potentially incompatible with Tcl versions on other platforms). An initial AndroWish SDK is provided which simplifies packaging of user defined trimmed down APKs (Android packages). The tclepeg package has been added to make JPEG files into thumbnails. Some other packages are updated to newer versions: SQLite 3.9.1, BWidget 1.9.9, and RAL 0.11.7.
Something wicked this way comes	2015-08-22	This release adds various new "borg" minor commands, e.g. to read images from the device camera(s) into Tk photo images, VecTcl 0.2, and interfaces to the ZBar and libdmtx barcode scanners. SQLite is updated to version 3.8.11.1. Subpackages were updated to newer upstream versions, and many bugs were fixed.
The Blues Brothers	2015-06-16	This release adds a Muzic compatible MIDI sound package to AndroWish. SQLite is updated to version 3.8.10.2. The tkpath widget and Bluetooth Low Energy module have been improved. Bugs in the Xlib emulation have been fixed. The "borg" command supports the new minor commands "trace" and "brightness".
Don Quixote	2015-04-23	This release adds support for Bluetooth Low Energy (aka Bluetooth Smart or Bluetooth 4.0) and the tkpath widget to AndroWish. SQLite is updated to version 3.8.9. Many bugs in the Xlib emulation and in the ZIP virtual filesystem have been fixed. TrueType font rendering speed for Tk widgets is improved. The "borg" command supports the new minor commands "broadcast", "providerinfo", and "queryconsts".
Pi Day	2015-03-14	This is mainly a bug fix and update release. Many little annoying problems in the Xlib emulation and in AndroWish startup were fixed: transient window handling, mouse/touch coordinate translation, loading of supplemental Java classes for Bluetooth and USB, etc. Many packages are updated to newer versions: Tcl/Tk 8.6.4, SQLite 3.8.8.3, tls 1.6.4, TclWS 2.3.8, itcl 4.0.3, itk 4.0.1, OpenSSL 1.0.1l, curl 7.41.0, tdbc* 1.0.3, and thread 2.7.2. The "borg" command has some new minor commands "systemproperties" and "phoneinfo" plus phone related virtual events.
Groundhog Day	2015-02-02	This release adds expect 5.45.2 and support for joysticks/game controllers to AndroWish. Many packages are updated: SQLite 3.8.8.2 including the ICU extension, nsf 2.0.0, tkimg 1.4.3, tls 1.6.4, trofs 0.4.8, and tklib to its latest upstream version. Some bugs in the X11 emulation are fixed: listbox selection, font metrics, dashed lines, various crashes. The built in ZIP file system is improved and fixes long standing issues with glob -directory.
Peter Pan	2014-12-29	This is mainly a bug fix release which improves stability of the 3d canvas widget, fixes "wm attributes -fullscreen" and on-screen keyboard handling, and adds proper support of the "sdltk screensaver" command. SQLite is updated to version 3.8.7.4, and the mentry and tablelist packages to their latest upstream versions. A separately packaged TkChat for Android is available which needs an installed AndroWish on the device.
All things are full of fools	2014-12-07	Stultorum plena sunt omnia (Marcus Tullius Cicero). Tcl and Tk are updated to version 8.6.3, SQLite to version 3.8.7.3. The brand new feature: DRH's 3d canvas widget now runs on Android! It uses an OpenGL to OpenGLES 1.1 emulation layer and renders to an off-screen texture which later is copied into the framebuffer and displayed by SDL which on most modern tablets/smartphones uses OpenGLES 2 for this task. YMMV in terms of stability of the 3d canvas depending on the quality of the device vendor's OpenGLES implementation.
The Gunpowder Plot	2014-11-05	Remember, remember AndroWish's first anniversary. The root window now can be zoomed and panned with two and three fingers, respectively. SQLite is updated to 3.8.7.1 and threading support is more stable.
The Flintstones	2014-09-30	Yabba dabba doo! The most prominent new feature is threading support in the X11 emulation layer allowing "package require Tk" from a Tcl thread. The tclral package is updated to version 0.11.2.

The Great Moon Hoax	2014-09-16	Lunar features: Tcl/Tk updated to version 8.6.2. Packages tclxml, snack, tclws, tclsoap, and vu widgets added. New virtual events and commands to deal with GPS/NMEA and tethering information. Drawing/rendering now performed in RGB888 instead of RGB565.
The Wizard of Oz	2014-08-17	Behind the curtain: SQLite updated to 3.8.6, OpenSSL updated to 1.0.1h, many icons now take the screen's pixel density into account, accelerometer and magnetic field sensors now report proper orientation data.
Alice In Wonderland	2014-07-28	Initial fossil import. Follow the white rabbit.

rfcomm command

rfcomm command

Name

rfcomm - transfer data over Bluetooth serial port profile; akin to the Tcl `socket` command.

Synopsis

```
package require Rfcomm
rfcomm ?-myaddr addr? ?-myport myport? ?-async? host port
rfcomm -server command ?-myaddr addr? port
```

Description

This command is used to obtain a channel which is able to transfer data over Bluetooth's serial port profile (SPP or SP). The arguments are nearly identical to the Tcl `socket` command. It returns a client or server channel handle. Client channels may be used with `gets`, `read`, `puts`, `fconfigure`, and `close`. Server channels return a new client channel in the *`command`* callback when an incoming connection was established.

For client channels (first command form), the *`host`* parameter must be given as a one- or two-element list: the first element is the Bluetooth address of the remote device, and the (optional) second element is the UUID of the remote service. If omitted the standard Bluetooth UUID for the Serial Port Profile `00001101-0000-1000-8000-00805F9B34FB` is used. The non-blocking connection mode (`-async` specified) uses readability of the channel to indicate connection state. This is different to normal `socket` channels, where writability provides this information. On Android, the local address of the client socket specified in the optional *`addr`* parameter is ignored.

For server channels, the first element of the *`addr`* parameter is the UUID for the local SDP (Service Discovery Protocol) record, i.e. the application identifier, and the optional second element is the friendly name of the service as advertised over SDP. On some Android versions the friendly name may not be an empty string, otherwise incoming connection requests are not fulfilled. The *`port`* parameter is usually ignored and should be specified as 0.

sdltk command

sdltk command

Name

sdltk - exposure of the SDL2 (Simple DirectMedia Layer) API.

Synopsis

`sdltk` *`option ?arg ...?`*

Description

This command is used to control portions of the Android (or Windows or Linux) system that the SDL2 framework exposes. Actual data processing for this framework is achieved by having handlers for virtual events.

`sdltk powerinfo`

> Returns a list of key-value pairs describing the state of the battery. The keys are `state`, `seconds`, and `percent`. The possible values for the state are `onbattery`, `nobattery`, `charging`, `charged`, and `unknown`. The other items are reported as integer numbers.

`sdltk accelerometer on|off`

> Turns event reporting of the device's accelerometer on or off. Creates top-level virtual events `<<Accelerometer>>` when turned on. This command is not usable on Windows and Linux.

`sdltk accelbuffer` *`axis`*

> Returns the accelerometer values for *`axis`* (1..3) which have been read during the last second as a list of integer values in the range -32768 .. 32767. The time resolution is identical with the framerate (20 ms). The values can be read out anytime independent of the accelerometer event enable state. The buffer is filled based on occurrences of the `<<Accelerometer>>` virtual event, missed values with respect to the framerate are interpolated. This command is not usable on Windows and Linux.

`sdltk textinput ?on|off ?`*`x y ?hint`*`???`

> Returns the state of the virtual keyboard or switches the virtual keyboard on or off. The optional coordinate pair is a hint for the system where the insertion cursor is displayed in screen coordinates. This allows the system to adjust the application's screen in order to display the insertion cursor when the virtual keyboard is active. The `entry`, `ttk::entry`, `text`, and `spinbox` widgets have standard bindings which activate text input on left mouse button press (or equivalent touch event) if the widget's state is not disabled. Activation of text input for these widgets can be turned off entirely by providing a dummy bindtag named `SdlTkNoTextInput`. Android specific: the *`hint`* parameter is an integer which controls the kind of virtual keyboard to be displayed. Known values are 0 (normal keyboard), 2 (number input), 3 (phone number input), 4 (date/time input).

`sdltk android`

> Returns true when running on Android, false otherwise, i.e. when built for Windows or Linux platforms.

`sdltk framebuffer`

> Returns true when the video driver resembles a framebuffer, i.e. no windowing manager is available. Currently this is the case for Android, the Raspberry Pi video driver (RPI), the Linux KMSDRM video driver, and the jsmpeg video driver.

`sdltk isandroidtv`

> Returns true when running on an AndroidTV device (currently untested).

`sdltk ischromebook`

> Returns true when running on a Chromebook (currently untested).

`sdltk maxroot`

> Returns the maximum size of the root window as two element list made up of width and height in pixels. The maximum size is device dependent and determined by the maximum texture size of the underlying OpenGL/OpenGLES drivers.

`sdltk root ?`*`width height`*`?`

> When invoked without *`width`* and *`height`* parameters the command returns the current size of the root window as two element list of integers. When *`width`* and *`height`* are given, the root window is resized to

the size given. When both *width* and *height* are given as zero, the root window is resized to the device screen size.

`sdltk vsync`

Waits until the next screen refresh and returns the number of screen refreshes which happened during that wait. The maximum wait time is limited to 20 milliseconds (the internal tick rate for screen updates) but can be longer due to system load.

`sdltk viewport ?x y ?width height??`

Changes the viewport (root window to device screen) to allow zooming and panning of the root window. When invoked without parameters, the current viewport settings are returned as a four element list of integers. When the *x* and *y* parameters are given, the viewport is shifted that *x* and *y* are shown in the top-left corner of the screen. When all four parameters are given, the viewport is adjusted accordingly, i.e. *width* and *height* determine the zoom factor, and *x* and *y* the top-left corner of the view. Note however, that the aspect ratio is retained, i.e. the given parameters are adjusted to keep the aspect.

`sdltk touchtranslate ?mask?`

Controls touchscreen event translation, or reports the current translation state. *mask* is a bit mask controlling various translations. Bit 0 (mask 1) turns on translation of middle/right mouse buttons, i.e. fast wipes with one finger are translated to mouse button 2 press/motion/release events to allow scrolling of listboxes, entries, and text widgets. Slow wipes still deliver mouse button 1 motion events. Holding down one finger for about a second is translated into mouse button 3 press for context menus. Bit 1 (mask 2) turns on pinch-to-zoom with two fingers which is reported as a virtual event named `<<PinchToZoom>>`. Bit 2 (mask 4) turns on pinch-to-zoom and wipes for zooming and panning the root window. When both, bits 1 and 2 are on (mask equals 6), zooming the root window requires three instead of two fingers and panning four instead of three fingers. Bit 3 (mask 8) turns on translation of finger events to the current viewport settings, i.e. the `<<FingerUp>>`, `<<FingerDown>>`, and `<<FingerMotion>>` events are translated to the current viewable portion of the root window instead of the device screen. Bit 4 (mask 16) turns on reporting of finger down/up events for up to 10 fingers as `<ButtonPress>` and `<ButtonRelease>` events with button numbers 10 to 19. However, no provisions are taken to ensure proper implicit button grabs like a real X server would do, thus use this feature with caution. The default touchscreen translation mode on startup is mask 13 (bits 0, 2, and 3 are on), i.e. everything except `<<PinchToZoom>>` and finger down/up as `<ButtonPress>`/`<ButtonRelease>` is enabled. On Windows and Linux platforms only bit 3 (mask 8) to control the viewport is supported.

`sdltk screensaver ?on|off?`

Turns the screen saver on or off or reports the current state of the screensaver.

`sdltk joystick ids`

Returns a list made up joystick ids (in SDL2 referred to as joystick instance identifiers) which are reported in related virtual events. These ids are integer numbers which increase for each new detected joystick.

`sdltk joystick name id`

Returns the name of the joystick identified by *id*.

`sdltk joystick guid id`

Returns the globally unique id (GUID, 128 bit string) of the joystick identified by *id*.

`sdltk joystick numaxes id`

Returns the number of axes of the joystick identified by *id*.

`sdltk joystick numballs id`

Returns the number of balls of the joystick identified by *id*.

`sdltk joystick numbuttons id`

Returns the number of buttons of the joystick identified by *id*.

`sdltk joystick numhats id`

Returns the number of hats of the joystick identified by *id*.

`sdltk addfont filename`

Adds TrueType font(s) contained in *filename* and returns the font family names which were added. If the font already has been loaded an error is thrown.

`sdltk hasgl`

Returns true when OpenGL support is available, e.g. for the 3D canvas widget.

`sdltk log priority message`

Outputs the log message *message* using SDL's logging facility. *priority* specifies the priority of the log message and must be one of `verbose`, `debug`, `info`, `warn`, `error`, or `fatal` (from lowest to highest).

`sdltk deiconify`

Deiconifies the SDL root window (not usable on Android and Wayland).

`sdltk fullscreen`

Makes the SDL root window into a fullscreen window (not usable on Android and Wayland). The SDL root window must be resizable (command line option `-sdlresizable`).

`sdltk iconify`

Iconifies (minimizes) the SDL root window (not usable on Android and Wayland).

`sdltk maximize`

Maximizes the SDL root window (not usable on Android and Wayland). The SDL root window must be resizable (command line option `-sdlresizable`).

`sdltk restore`

Restores the last unmaximized geometry of the SDL root window (not usable on Android and Wayland).

`sdltk withdraw`

Withdraw (hides entirely) the SDL root window (not usable on Android and Wayland).

`sdltk opacity` *value*

Query or set the opacity of the SDL root window. *value* must be a floating point number between 0.0 and 1.0 (not usable on Android). On POSIX operating systems the window manager must support transparent toplevels for this setting having an effect.

`sdltk fonts`

Returns a list made up of font information in the form of three elements XLFD, file name, font index of all registered fonts.

`sdltk vrmode` *?mode ?distortion rescale??*

Experimental VR headset mode currently only supported on the Android platform. If *mode* is specified, it changes the VR headset mode to one of the following: Mode 0 for normal operation, in mode 1 the root window is duplicated along its horizontal axis and scaled up or down, in mode 2 the root window must be managed as left and right halves by the application, and in mode 3 the root window is duplicated along its horizontal axis without scaling. For all modes except mode 0 touch screen panning and zooming on Android is turned off and touch coordinates in X are reported equal for both left and right halves of the screen. All modes except mode 0 turn on a shader performing a barrel distortion (when OpenGL ES 2 is available) which theoretically compensates the effect of lenses of a VR headset. The optional parameters *distortion* and *rescale*, if present, must be specified as floating point numbers and control the degree of distortion. In order to flip the image(s) horizontally and/or vertically, *mode* can be bitwise or'ed with 4 (horizontal flip) and/or 8 (vertical flip) for all modes except 0. If *mode* and additional arguments are omitted, the currently active mode including the distortion control parameters are returned as a Tcl list of three elements.

`sdltk pointer` *?flag?*

Queries or sets the state of the mouse pointer shape. If present, *flag* must be a boolean value and specifies the new state. If not present, the current state is returned as 0 (off) or 1 (on).

`sdltk touchcalibration` *?xmin xmax ymin ymax swapxy?*

Queries or sets the calibration data for resistive touchscreens supported on certain SDL video drivers (currently Linux EVDEV devices with KMSDRM or RPI video drivers). The calibration data consists of five integer numbers which are returned as a list, when the command is called without parameters.

`sdltk size` *?width height?*

Queries the size of the enclosing SDL root window when *width* and *height* parameters are omitted. A two element list is returned with the current width and height in pixels. If parameters are given, the enclosing SDL root window is resized respectively, provided that the command line parameter `-sdlresizeable` was specified and the command line parameter `-sdlfullscreen` was not specified on startup. However, changing the SDL root window size is not supported on framebuffer like devices (see `sdltk framebuffer`).

Touchscreen and Accelerometer Events

Using the sdltk framework usually requires liberal use of virtual event handlers. The virtual events include:

`<<Accelerometer>>`

Event associated with the accelerometer (activated with `sdltk accelerometer on`). `%s` is substituted with the accelerometer axis {1..3} and `%x` with the accelerometer value in the range {-32768...+32767}. This event is reported to toplevel widgets only.

`<<FingerDown>>`

A touch event.

`<<FingerUp>>`

A touch completion event.

`<<FingerMotion>>`

A touch movement (sliding) event. The fields `%x` and `%y` are substituted with the finger position scaled to {0...9999} of the device screen or viewport, `%X` and `%Y` with the motion difference scaled to {-9999...+9999}, `%t` with the pressure scaled to {0...9999}, and `%s` with the finger identifier {1...10}. These substitutions are performed for all finger related touch events.

`<<PinchToZoom>>`

A zoom gesture event. `%X` and `%Y` are substituted with the root window coordinate of the center of the two fingers, `%x` with the distance between the two fingers, and `%y` with the angle measured in 64 times degrees CCW starting at 3 o'clock. The finger state is reported in the `%s` substitution as 0 (zoom motion), 1 (zoom start, i.e. 2nd finger down event), 2 (zoom end by 1st finger up event), 3 (zoom end by 2nd finger up event).

Joystick Events

Following virtual events are reported for joysticks and game controllers:

`<<JoystickAdded>>`, `<<JoystickRemoved>>`

Event generated when a joystick or game controller is plugged or unplugged. The field `%X` is substituted with the joystick id (instance identifier in SDL2 terminology).

`<<JoystickMotion>>`

Similar to `<<Accelerometer>>` this event is reported when the position of the joystick has changed. An additional substitution is made for `%X` which receives the joystick id (instance identifier in SDL2 terminology).

`<<TrackballMotion>>`

A joystick trackball has moved. The fields `%x` and `%y` are substituted with the deltas of the move, `%s` with the trackball number counted from 1, the field `%X` indicates the joystick id.

`<<HatPosition>>`

A joystick hat has changed. The field `%x` is substituted with the value of the hat, `%s` with the hat number counted from 1, the field `%X` indicates the joystick id.

`<<JoystickButtonUp>>`, `<<JoystickButtonDown>>`

A joystick button was pressed or released. The field `%s` is substituted with the button number counted from 1, the field `%X` indicates the joystick id.

Events related to the device screen

`<<ViewportUpdate>>`

This event is sent to toplevel widgets when the viewport has changed. `%x` and `%y` are substituted with the viewport offset (top-left corner of the screen), `%X` and `%Y` with the width and height, respectively, and `%s` with the scale factor (relation of root window size to displayed size) scaled to 10000.

Events related to the app life-cycle

These events are direct translations from SDL events (**SDL_APP_*** in SDL header files) and depend on platform support. They are reported to toplevel widgets only.

`<<LowMemory>>`

System is in low memory situation. Although implemented for Android and iOS, this event was never observed in reality.

`<<Terminating>>`

App is terminating. Although implemented for Android and iOS, this event was never observed in reality, maybe due to timing regarding threads.

`<<WillEnterBackground>>`

App's screen will be put in background.

`<<DidEnterBackground>>`

App's screen is in the background.

`<<WillEnterForeground>>`

App's screen will be put in foreground. On Android, not reported on startup of the app.

`<<DidEnterForeground>>`

App's screen is in the foreground. On Android, not reported on startup of the app.

Note that on Android the system may kill an app at any time due to low memory situations. In order to keep some app state persistent, the best option is to record each change immediately. Another option is using the `<<WillEnterBackground>>` virtual event since it may be received before unexpected app termination.

Accelerometer Example

```
proc showaccel {canvas axis value} {
    set ix 0
    set iy 0
    if {$axis == 1} {
        set ix [expr {$value / 256}]
    } elseif {$axis == 2} {
        set iy [expr {$value / 256}]
    } elseif {$axis == 3} {
        set ::pos(t) [expr {($value / 256) % 360}]
    } else {
        return
    }
    if {![info exists ::pos(x)]} {
        set ::pos(x) [expr [winfo width $canvas] / 4]
        set ::pos(y) [expr [winfo height $canvas] / 4]
        set ::pos(t) 0
    }
    set ::pos(x) [expr {$::pos(x) + $ix}]
    set ::pos(y) [expr {$::pos(y) + $iy}]
    if {$::pos(x) < 50} {
        set ::pos(x) 50
    } elseif {$::pos(x) > [winfo width $canvas] - 50} {
        set ::pos(x) [expr {[winfo width $canvas] - 50}]
    }
    if {$::pos(y) < 50} {
        set ::pos(y) 50
    } elseif {$::pos(y) > [winfo height $canvas] - 50} {
        set ::pos(y) [expr {[winfo height $canvas] - 50}]
    }
    if {$axis == 3} {
        $canvas delete a
        set x0 [expr {$::pos(x) - 48}]
        set x1 [expr {$x0 + 96}]
        set y0 [expr {$::pos(y) - 48}]
        set y1 [expr {$y0 + 96}]
        $canvas create arc $x0 $y0 $x1 $y1 -fill yellow -outline red \
            -width 6 -start [expr {330 - $::pos(t)}] -extent -300.0 -tags a
    }
}

wm attributes . -fullscreen 1
canvas .c -bg black -bd 0 -highlightthickness 0
pack .c -side top -fill both -expand 1 -padx 0 -pady 0
set f [open [info script]]
.c create text 20 120 -anchor nw -tag s -font {Courier 5} -text [read $f] \
    -fill gray50
close $f
button .c.x -text Exit -command {exit 0}
.c create window 30 60 -anchor nw -tag x -window .c.x
bind . <<Accelerometer>> {showaccel .c %s %x}
sdltk accelerometer on
```

Pinch-to-zoom Example

```
proc showzoom {canvas rootx rooty dist angle state} {
    $canvas itemconf t -text "XY: $rootx,$rooty L: $dist P: $angle S: $state"
    $canvas delete a
    # state 0 -> zoom motion
    # state 1 -> zoom start
    # state 2 -> zoom end, 1st finger up
    # state 3 -> zoom end, 2nd finger up
    if {$state < 2} {
```

```
            set phi [expr {$angle / 64.0}]
            set x0 [expr {$rootx - [winfo rootx $canvas] - $dist / 2}]
            set x1 [expr {$x0 + $dist}]
            set y0 [expr {$rooty - [winfo rooty $canvas] - $dist / 2}]
            set y1 [expr {$y0 + $dist}]
            $canvas create arc $x0 $y0 $x1 $y1 -fill yellow -outline red -width 6 \
                -start [expr {330 - $phi}] -extent -300.0 -tags a
        }
    }

    wm attributes . -fullscreen 1
    sdltk touchtranslate 15 ;# turn <<PinchToZoom>> on
    canvas .c -bg black -bd 0 -highlightthickness 0
    pack .c -side top -fill both -expand 1 -padx 0 -pady 0
    set f [open [info script]]
    .c create text 30 120 -anchor nw -tag s -font {Courier 6} -text [read $f] \
        -fill gray50
    close $f
    .c create text 30 30 -anchor w -fill green -tag t -font {Helvetica 16} \
        -text "Try pinch-to-zoom with two fingers"
    button .c.x -text Exit -command {exit 0}
    .c create window 30 60 -anchor nw -tag x -window .c.x
    bind .c <<PinchToZoom>> {showzoom %W %X %Y %x %y %s}
```

Disable Android keyboard input to a text widget

```
bindtags .mywidget [list SdlTkNoTextInput {*}[bindtags .mywidget]]
```

snap7 command

Name

snap7 - Tcl interface to the Snap7 library

Synopsis

```
package require Tcl 8.6
package require snap7
snap7::new cmd
cmd destroy
cmd connect addr port rack slot
cmd disconnect
cmd conntype type
cmd param ?name? ?value?
cmd isconnected
cmd pdulength
cmd dbread db start count
cmd dbreada db start count
cmd dbwrite db start data ...
cmd dbwritea db start bytes
```

Description

This package provides a Tcl interface to the Snap7 library (see http://snap7.sourceforge.net/) using Ffidl and TclOO.

Commands

`snap7::new` *cmd*

Creates a new command *cmd* which implements a Snap7 connection object. Further operations on that object are carried out by invoking methods on *cmd*.

cmd `destroy`

Destroys the connection object *cmd*, releases resources and closes communications links.

cmd `connect` *addr port rack slot*

Connects the connection object *cmd* to its peer using the IP address *addr*, the TCP port number *port* and further address information (*rack* and *slot* numbers).

cmd `disconnect`

Closes the connection of the connection object *cmd*.

cmd `conntype` *type*

Sets the connection type of the connection object *cmd*. Must be called before a connection is made using the `connect` method. Valid values for *type* are 1 (PG), 2 (OP), and 3 (basic).

cmd `param` *?name? ?value?*

If invoked without arguments, returns a list of parameter names which can be queried or set on the connection object *cmd*. If *name* is provided, a query of this named parameter is performed. If both, *name* and *value* are provided, the named parameter is set to the value given.

cmd `isconnected`

Returns true or false depending on connection state of the connection object *cmd*.

cmd `pdulength`

Returns a two element list made up of requested and negotiated PDU length of the connection object *cmd*.

cmd `dbread` *db start count*

Reads *count* bytes beginning at *start* from the data block *db* using the connection object *cmd*. Data is returned as a list of integer numbers.

cmd `dbreada` *db start count*

Reads *count* bytes beginning at *start* from the data block *db* using the connection object *cmd*. Data is

returned as a byte array.

cmd dbwrite *db start data ...*

Writes the numbers specified by *data* and following arguments as bytes beginning at *start* into the data block *db* using the connection object *cmd*.

cmd dbwritea *db start bytes*

Writes the byte array *bytes* beginning at *start* into the data block *db* using the connection object *cmd*.

tclcan

can command

Name

can - Tcl interface to Linux SocketCAN

Synopsis

```
package require Tcl 8.6
package require tclcan
can bcmopen ifname
can bitrate ifname ?rate? ?sample_point?
can bittiming ifname
can bitttiming_const ifname
can berr ifname
can clock ifname
can close chan
can ctrlmode ifname ?mode ...?
can devstat ifname
can dump chan
can interfaces
can open ifname
can read chan
can restart ifname
can restart_ms ifname ?ms?
can start ifname
can state ifname
can stop ifname
can write chan canid data ?ifindex?
can write chan opcode flags count time1 time2 canid ?ifindex ...?
```

Description

This package provides Tcl support for Linux SocketCAN **CAN_RAW** and **CAN_BCM** socket types. The package implements a new channel type and a Tcl command to perform operations on these channels. The standard `gets`, `puts`, and `read` Tcl commands are not supported, but `close`, `fconfigure`, and `fileevent` are available as for normal channels, e.g. sockets. When the **libsocketcan** shared library is available, various subcommands can be used to manage CAN interfaces, too.

Commands

`can bcmopen` *`ifname`*

> Opens a channel by creating a broadcast manager socket (type **CAN_BCM**) on the given CAN interface *`ifname`*. If *`ifname`* is specified as an empty string, the channel is bound to all CAN interfaces. The command returns an identifier for the channel which is to be used in subsequent `can read` and `can write` commands.

`can bitrate` *`ifname ?rate? ?sample_point?`*

> Gets or sets the bitrate *`rate`* (and sets optional sample point to *`sample_point`*) on the CAN interface *`ifname`*.

`can bittiming` *`ifname`*

> Retrieves the current bit timing of the CAN interface *`ifname`*. For details refer to **/usr/include/can_netlink.h**.

`can bittiming_const` *`ifname`*

> Retrieves configuration on bit timing of the CAN interface *`ifname`*. For details refer to **/usr/include/can_netlink.h**.

`can berr` *`ifname`*

> Retrieves error counters of the CAN interface *`ifname`*. The result is a dictionary made up of the keys `txerr` and `rxerr` with respective integer error counters.

`can clock` *`ifname`*

> Retrieves the clock frequency of the CAN interface *`ifname`*. For details refer to **/usr/include/can_netlink.h**.

`can close` *`chan`*

Closes the channel *chan* which was formerly obtained by `can open`. This is equivalent to invoking the `close` command with *chan* as parameter.

`can ctrlmode` *ifname ?mode ...?*

Gets or sets modes on the CAN interface *ifname*. If no *mode* is specified, the current active modes are returned as a list. Otherwise, *mode* must be one or more words of `loopback`, `listenonly`, `3_samples`, `one_shot`, `berr_reporting`, `fd`, and `presume_ack`. In order to turn a mode off, prefix the word with a minus sign. Likewise, to turn it on, a plus sign may be optionally used as prefix.

`can devstats` *ifname*

Retrieves device statistics as a dictionary. For details refer to **/usr/include/libsocketcan.h** and **/usr/include/can_netlink.h**.

`can dump` *chan*

Reads a **CAN_RAW** or **CAN_BCM** message off *chan* and returns a formatted representation of it as a list. The list is empty if no CAN message was pending on *chan*.

Otherwise, for **CAN_RAW** channels the list has five or six elements which are: 1. an integer time stamp equivalent to `clock microseconds`, 2. the interface index (see `can interfaces`), 3. the CAN identifier as a hexadecimal string with 0x prefix, 4. a frame format tag of the CAN message as `EFF` (extended frame format) or `SFF` (standard frame format) optionally followed by `|RTR` (remote transmission request) or `|ERR` (error frame), 5. the data length as a decimal number, and optionally 6. the data portion of the CAN message as hexadecimal dump without blanks and prefix.

For **CAN_BCM** channels the list is made up of: 1. an integer time stamp equivalent to `clock microseconds`, 2. the interface index (see `can interfaces`), 3. the major CAN identifier as a hexadecimal string with 0x prefix, 4. a frame format tag as described above, 5. the `BCM` opcode as one of `TX_STATUS`, `TX_EXPIRED`, `RX_STATUS`, `RX_TIMEOUT`, or `RX_CHANGED`, 6. the `BCM` flags separated by vertical bars (`SETTIMER`, `STARTTIMER`, `TX_COUNTEVT`, `TX_ANNOUNCE`, `TX_CP_CAN_ID`, `RX_FILTER_ID`, `RX_CHECK_DLC`, `RX_NO_AUTOTIMER`, `TX_RESET_MULTI_IDX`, and `RX_RTR_FRAME`), 7. the `BCM` count field, 8. the first `BCM` interval field as floating point number, 9. the second `BCM` interval field as floating point number, optionally 10. to 13. describing the first CAN frame as CAN identifier (hexadecimal string), the frame format tag (`EFF`, `SFF`, etc.), the data length, and the payload as hexadecimal dump. Fields 10. to 13. repeat for the respective number of CAN frames contained in the `BCM` message.

`can interfaces`

Returns a list of CAN network interface names and indices suitable for `can open`, `can read`, `can write` and link management subcommands.

`can open` *ifname*

Opens a channel (raw **AF_CAN** socket) on the given CAN interface *ifname*. If *ifname* is specified as an empty string, the channel is bound to all CAN interfaces. The command returns an identifier for the channel which is to be used in subsequent `can read` and `can write` commands.

`can read` *chan*

Reads a **CAN_RAW** or **CAN_BCM** message off *chan* as a list. The list is empty if no CAN message was pending on *chan*.

Otherwise, for **CAN_RAW** channels it is made up of four elements, 1. the CAN identifier as an integer number including flags as explained below, 2. the data portion of the CAN message as a byte array, 3. the interface index of the CAN interface the CAN message was received from, and 4. a boolean value indicating if more CAN messages can be read using `can read`.

For **CAN_BCM** channels it is made up of at least seven elements: 1. the interface index, 2. the major CAN identifier (see above), 3. the `BCM` operation as one of `TX_STATUS`, `TX_EXPIRED`, `RX_STATUS`, `RX_TIMEOUT`, or `RX_CHANGED`, 4. the `BCM` flags as an integer number, 5. the `BCM` count field as an integer number, 6. the first `BCM` interval field as a floating point number, and 7. the second `BCM` interval field as a floating point number. When CAN frames are part of the `BCM` message, each frame is a pair of CAN identifier as integer number and the payload as byte array of length 0 to 8 for normal frames or an integer number for RTR frames.

`can restart` *ifname*

Performs a link restart on the CAN interface *ifname*.

`can restart_ms` *ifname ?ms?*

Gets or sets the restart timer of the CAN interface *ifname*. *ms* must be specified as positive integer number of milliseconds.

`can start` *ifname*

Performs a link startup on the CAN interface *ifname*.

`can state` *ifname*

Retrieves the linmk state of the CAN interface *ifname*. The result is one of error_active, error_warning, error_passive, bus_off, stopped, sleeping, or unknown.

can stop *ifname*

Performs a link stop on the CAN interface *ifname*.

can write *chan canid data ?ifindex?*

Writes a **CAN_RAW** message to *chan*. *canid* is the CAN identifier as integer number, *data* a byte array of the data to be sent. The optional *ifindex* is the CAN interface index (see can interfaces) on which the message is to be sent. It is mandatory to specify *ifindex* when *chan* is bound to all interfaces, i.e. the interface name on can open was an empty string.

can write *chan opcode flags count time1 time2 canid ?ifindex ...?*

Writes a **CAN_BCM** message to *chan*. *opcode* must be a BCM operation out of the set TX_SETUP, TX_DELETE, TX_READ, TX_SEND, RX_SETUP, RX_DELETE, and RX_READ. *flags* must be a list with zero or more elements of the set SETTIMER, STARTTIMER, TX_COUNTEVT, TX_ANNOUNCE, TX_CP_CAN_ID, RX_FILTER_ID, RX_CHECK_DLC, RX_NO_AUTOTIMER, RX_ANNOUNCE_RESUME, TX_RESET_MULTI_IDX, and RX_RTR_FRAME. *count* is the counter for the first interval *time1*. The intervals *time1* and *time2* must be given as floating point numbers of seconds. *canid* is the major CAN identifier for the BCM message. *ifindex* is the interface index which is required, if the **CAN_BCM** channel was bound to all interfaces. All following optional arguments make up CAN frames and must be pairs of a CAN identifier and a byte array of 0 up to 8 bytes for normal frames, or an integer as data length for RTR frames.

CAN Identifiers

The Linux SocketCAN interface defines special bits in CAN identifiers which are made up of the three most significant bits in a 32 bit integer: 0x80000000 for extended frame format (EFF), 0x40000000 for remote transmission request (RTR), and 0x20000000 for error frames (ERR). The lower 29 (for EFF) or 11 (for SFF) bits make up the CAN identifier. In order to retrieve the real CAN identifier of a received CAN message from can read a binary and with the masks 0x1FFFFFFF or 0x7FF must be carried out. In order to send an RTR message, the CAN identifier must be binary or-ed with 0x40000000 for can write. In order to send a 29 bit CAN identifier it must be or-ed with 0x80000000.

Channel Options

The following list describes the additional channel options of CAN channels.

-error

The last system error message on the channel. This is a read-only option.

-filter *?list?*

Message filters applied on reception. *list* must be made up of an even number of integers specifying CAN identifiers and masks. The default is no filtering, expressed as two zero values. Up to 16 filters can be specified. For details refer to **/usr/include/linux/can.h**.

-loopback *?bool?*

Messages sent are looped back on the local system when enabled (on by default).

-ownmsgs *?bool?*

Messages sent are received on the same channel when enabled (off by default).

Link Management

The link management subcommands bitrate, bittiming, bittiming_const, berr, clock, ctlrmode, devstat, restart, restart_ms, start, state, and stop depend on an installed **libsocketcan** shared library for proper operation. Otherwise they report "function not implemented". All changes of link state by these commands usually require administrative rights. Either the calling process must have super user privileges or the **CAP_NET_ADMIN** capability must be effective. The latter can be achieved by a command similar to:

```
    setcap cap_net_admin+eip binary-package-requiring-tclcan
```

Furthermore, retrieving link information depends on CAN driver support. Usually, the virtual CAN driver **vcan** and drivers attached through a serial line discipline (using the **slcan_attach** or **slcand** programs) only provide rudimentary link state information.

Broadcast Manager Examples

Open BCM channel:

```
    set chan [can bcmopen can0]
```

Schedule sending the pattern 0x41424344 on CAN identifier 0x123 once per second:

```
    can write $chan TX_SETUP \
        {SETTIMER STARTTIMER} \
        0 0.0 1.0 0x123 0x123 ABCD
```

Dispatch receiving CAN identifier 0x123 with update rate limited to two seconds:

```
    can write $chan RX_SETUP \
        {SETTIMER RX_FILTER_ID RX_ANNOUNCE_RESUME} \
        0 0.0 2.0 0x123
```

Dump all received `BCM` messages on standard output:

```
    proc dump chan {puts [can dump $chan]}
    fileevent $chan readable [list dump $chan]
```

Test and debug strategies on AndroWish

Test and debug strategies on AndroWish

For interactive testing, follow the directions given in tkconclient.

When scripts are not run interactively but started using e.g. an icon on the Android home screen, script errors may show up in the Android system log buffer when not reported through the Tcl background error mechanism. In this case, the Android Debug Bridge (adb) should be used on a development system. Refer to the description of the logcat command-line tool and see an example output in the last image of the AndroWish SDK documentation.

Similarly, when explicit log output shall be written by application code, the `borg log ...` command or the `sdltk log ...` command can be used.

Output to the `stderr` and `stdout` channels in non-interactive scripts is normally not shown, but can be easily displayed, too, when the console window is made viewable using `console show`.

tkconclient

tkconclient

`tkconclient` is described in the Tcl Wiki as means for remote access to another Tcl interpreter using the `tkcon` console in socket mode.

For an interactive AndroWish this can be achieved by adding these lines to `~/.wishrc` (the Tcl script getting sourced when an interactive `wish` or AndroWish is started)

```
package require tkconclient
tkconclient::start 12345
```

meaning that TCP port 12345 is accepting incoming connections from `tkcon` on all interfaces. If the Android device is connected to the development system using an USB cable, it is possible to redirect port 12345 to that USB connection:

```
# on development system, instruct adb (Android Debug Bridge from SDK)
# to forward TCP port 12345
adb forward tcp:12345 tcp:12345
```

Then `tkcon` can connect in socket mode to `localhost:12345`. Alternatively, the netcat tool `nc` can be used but no input prompts are shown:

```
# netcat on development system, either called "netcat" or "nc"
nc localhost 12345
```

Alternatively, the `socat` tool can be used similar to netcat:

```
# socat on development system
socat TCP:localhost:12345 STDIO
```

Even ye good olde `telnet` should do:

```
# telnet on development system
telnet localhost 12345
```

Similarly, the comm package from tcllib can be used in `~/.wishrc` as

```
package require comm
comm::comm new comm::comm -port 12347 -local 1 -listen 1 -silent 1
```

where the TCP port used is 12347 on the local interface. The adb redirection in this case is:

```
adb forward tcp:12347 tcp:12347
```

My own `~/.wishrc` is somewhat larger:

```
# Start socket for tkcon
#
# When used over ADB USB debug connection
# the TCP port 12345 must be forwarded using
#
#   adb forward tcp:12345 tcp:12345

catch {
    package require tkconclient
    tkconclient::start 12345
}

# Start socket for comm
#
# When used over ADB USB debug connection
# the TCP port 12347 must be forwarded using
#
#   adb forward tcp:12347 tcp:12347

catch {
    package require comm
    comm::comm new comm::comm -port 12347 -local 1 -listen 1 -silent 1
}

# Start dropbear SSH/SFTP daemon using librun.so
# which is on the path of executable programs and
# located in the directory where all AndroWish
# shared libraries are installed.
```

```
    #
    # When used over ADB USB debug connection
    # the TCP port 12346 must be forwarded using
    #
    #   adb forward tcp:12346 tcp:12346
    #
    # The public key of the development system
    # must have been copied to $env(HOME)/.ssh/authorized_keys
    # of the Android device. $env(HOME) is usually /data/data/tk.tcl.wish/files
    #
    # This allows to SSH into the device as the AndroWish user
    # or to SFTP to/from the device as the AndroWish user.
    # That poor AndroWish user is the uid under which the Android
    # package manager decided to install the AndroWish APK.

    catch {
        exec librun.so libdropbear.so dropbear_main -R -p 12346
    }

    # Other goodies accessible through librun.so
    #
    # tclsh:        librun.so libtcl.so tclsh ...
    # sqlite3:      librun.so libtclsqlite3.so sqlite3_shell ...
    # ssh:          librun.so libdropbear.so cli_main ...
    # scp:          librun.so libdropbear.so scp_main ...
    # dropbearkey:  librun.so libdropbear.so dropbearkey_main ...
    # curl:         librun.so libcurl.so curl_main ...
```

opcua command

Name

opcua - Tcl binding to the OPC/UA implementation of **http://www.open62541.org**

Synopsis

```
package require topcua
opcua cmd ?arg?
```

Description

This command provides several operations to manage and communicate using the OPC/UA implementation of **http://www.open62541.org**. It is available on common POSIX and Windows platforms. *cmd* indicates which operation to carry out. Any unique abbreviation for *cmd* is acceptable. The valid commands are:

`opcua acl` *handle ?user pass ...?*

> Modifies the user/password based access control list of the server object *handle*. This command must be called after the server object has been created (see `opcua new server`) and before it is put into operation (see `opcua start`). To allow anonymous logins, specify an empty username in the arguments, whose password will be ignored.

`opcua add` *handle* `DataType` *nodeid parent reftype brname ?attrs?*

> Adds an new node of node class `DataType` in the object *handle* and returns the node identifer. The parameter *nodeid* is the requested new node identifier of the node to be created. *parent* is the parent node identifier and *reftype* the reference type or node identifier of the reference between the parent and the new node. *brname* is the browse name (see section **Qualified Names**) of the new node. The optional *attrs* parameter specifies attributes for the new node in form of a dictionary (see `opcua attr default`). If it is omitted, default values are used. The *DisplayName* attribute if left empty is preset to the name part of the browse name parameter.

`opcua add` *handle* `Method|SimpleMethod` *nodeid parent reftype outargs brname inargs cmd ?attrs?*

> Adds an new node of node class `Method` in the object *handle* and returns the node identifer. In the `SimpleMethod` command form the parameter *outargs* describes the output arguments of the method as a list of zero or more pairs of data type and argument names. To force an argument to be a scalar, the argument name must be prefixed with an exclamation mark. To force an argument to be an array, the argument name must be prefixed with an asterisk. Likewise, *inargs* describes the input arguments of the method with identical prefix rules. In the `Method` command form, both *outargs* and *inargs* must be provided as lists of dicts with the template obtained from `opcua types empty` *Argument*. The parameter *cmd* is the Tcl callback to handle the method invocation, see section **Method Callbacks** for more information. For the other parameters, refer to `opcua add DataType`.

`opcua add` *handle* `Namespace` *name*

> Adds the new namespace *name* to the server object *handle* and returns a numeric identifier for this namespace.

`opcua add` *handle* `Object` *nodeid parent reftype brname ?typeid attrs?*

> Adds an new node of node class `Object` in the object *handle* and returns the node identifer. The optional parameter *typeid* must be a known data type name (see `opcua types`) or a node identifier of a data type. For the other parameters, refer to `opcua add DataType`.

`opcua add` *handle* `ObjectType` *nodeid parent reftype brname ?attrs?*

> Adds an new node of node class `ObjectType` in the object *handle* and returns the node identifer. For the other parameters, refer to `opcua add DataType`.

`opcua add` *handle* `Reference` *srcid reftype target ?forward?*

> Adds a reference of type *reftype* (see `opcua reftype`) between the node identifiers *srcid* and <*target* on the object *handle*. The optional parameter *forward* must be a boolean indicating the direction of the reference (true, the default, is forward, false is inverse).

`opcua add` *handle* `ReferenceType` *nodeid parent reftype brname ?attrs?*

> Adds an new node of node class `ReferenceType` in the object *handle* and returns the node identifer. For the other parameters, refer to `opcua add DataType`.

`opcua add` *handle* `Variable` *nodeid parent reftype brname ?typeid attrs cmd?*

Adds an new node of node class `Variable` in the object *`handle`* and returns the node identifer. The optional parameter *`typeid`* must be a known data type name (see `opcua types`) or a node identifier of a data type or an empty string for a default value. Parameter *`cmd`* is an optional data source callback which produces (read operation) or consumes (write operation) the variable's value. See section **Data Source Callbacks** for more information. For the other parameters, refer to `opcua add DataType`.

`opcua add` *`handle`* `VariableType` *`nodeid parent reftype brname ?typeid attrs?`*

Adds an new node of node class `VariableType` in the object *`handle`* and returns the node identifer. The optional parameter *`typeid`* must be a known data type name (see `opcua types`) or a node identifier of a data type or an empty string for a default value. For the other parameters, refer to `opcua add DataType`.

`opcua add` *`handle`* `View` *`nodeid parent reftype brname ?attrs?`*

Adds an new node of node class `View` in the object *`handle`* and returns the node identifer. For the other parameters, refer to `opcua add DataType`.

`opcua appdesc` *`handle ?description?`*

Queries or sets the application description of the object *`handle`*. For a query, the current application description is returned as a a dictionary. This dictionary can be used as basis for modification and further to set a new application *`description`*. The set operation can be performed only on stopped server objects and unconnected client objects. Note, that the `ApplicationUri` component of the application description must match the corresponding information in certificates.

`opcua attrs ?list|default|numeric?` *`?name?`*

Without further parameters returns a list of attribute names the `opcua read` and `opcua write` commands support, e.g. `Value`, `NodeClass`, etc. With the `list` keyword a list of the data types used as attributes for creation of nodes with the `opcua add` command is returned. With the `default` keyword combined with the *`name`* of the data type a dictionary describing the default attributes of this type is returned, e.g `opcua attrs default` *`DataTypeAttributes`* yields a default dictionary for creation of a `DataType` node. With the *`numeric`* keyword combined with the *`name`* of the data type, the numeric value of the attribute is returned.

`opcua attr_init` *`name body ...`*

Helper function to create and return a dictionary for the attribute type *`name`* which is initially filled with defaults from `opcua attrs default...` and finally modified by the Tcl code in *`body`* with the same rules as in the `dict with` command. The parameters following *`body`* are assigned to the list *`args`* as in a `proc`. In contrast to `dict with`, *`body`* is executed in a call frame of its own.

`opcua browse` *`handle nodeid ?dir reftype mask ...?`*

Performs a browse operation on the client or server object *`handle`* starting at the node *`nodeid`*. The browse direction can be specified with the *`dir`* parameter as `Forward`, `Inverse`, or `Both`. Forward is the default direction. The optional *`reftype`* parameter selects the types of references to be considered. References can be preceeded with an exclamation mark in order to reverse their direction. A reference may be abbreviated as slash for `HierarchicalReferences` or as dot for `Aggregates`. If *`reftype`* is not specified, all nodes referenced to/from *`nodeid`* are reported. The optional *`mask`* and following parameters select specific node classes `Object`, `Variable`, `Method`, `ObjectType`, `VariableType`, `ReferenceType`, `DataType`, and `View`. The result of the browse operation is a list where each item is made up of the six elements node identifier, browse name (qualified name), display name (locale and text), node class, reference node identifier, and type node identifier.

`opcua call` *`handle nodeid methodid ?type value ...? ?`*`-async` *`cmd?`*

Calls the method with node identifier *`methodid`* on the object with node identifier *`nodeid`* on the client or server object `handle` with parameters described by pairs of *`type`* (data type, e.g. `Int32` or `String`) and value (the parameter's value). The method's result is returned. The method is carried out on the server, i.e. when directly used with a server handle there's no network traffic since the method is run locally. The *`type`* parameters should be prefixed with an asterisk or an exclamation mark in order to achieve the same semantic as in a method definition with `opcua add SimpleMethod`. Otherwise, array vs. scalar interpretion is automatically performed, i.e. when the corresponding *`value`* is a list, it is used as an array. For client objects, asynchronous operation is carried out when the last two parameters are `-async` and a (possibly empty) callback command *`cmd`*. In case of a non-empty callback the command returns an integer request identifier, which can be used to cancel the asynchronous operation. Otherwise the call is performed but the result is ignored. See section **Asynchronous Operations** for more information.

`opcua cancel` *`handle reqid ...`*

Cancels one or more asynchronous requests on the client object *`handle`*. The requests to be cancelled are identified by their integer identifiers *`reqid`*. The associated callbacks are evaluated with a timeout status code.

`opcua cert` *`handle cert pkey ?trust ...?`*

Loads the certificate *`cert`* and public key *`pkey`* into the client or server object *`handle`*. Both must be byte arrays. The optional parameters *`trust`* are zero or more byte arrays with certicates which are added to the object's trust list. If the underlying open62541 library does not support encryption, this command fails with an appropriate error message. If it succeeds, it forces a server object to only allow encrypted

sessions. Similarly, a client object tries to use a sign-and-encrypt endpoint of a server.

`opcua children` *`handle nodeid`*

Returns the child node identifiers of the given node identifier *`nodeid`* on the client or server object *`handle`*.

`opcua connect` *`handle url ?user password?`*

Connects the client object *`handle`* to the URL *`url`* using the optional credentials *`user`* and *`password`*.

`opcua connect` *`handle url`* `-async`

Connects the client object *`handle`* to the URL *`url`*. The operation is asynchronous, i.e. the connection establishment takes place in background. It can be observed with the optional `onclientstate` callback of the client object.

`opcua const` *`?name ...?`*

Without optional parameters returns a list of names for which mappings to numerical values are known. If *`name`* is provided, the numerical value for the name is returned. When more than one *`name`* is given, a bitwise OR of the values for the names is returned. *`name`* can be optionally prefixed with `UA_`, i.e. `ACCESSLEVELMASK_READ` and `UA_ACCESSLEVELMASK_READ` are mapped to the same numerical value.

`opcua createcert` *`fmt subjects subjectaltnames ?bits?`*

Creates a self signed certificate and public key given format *`fmt`* (`der` or `pem`), a list of *`subjects`*, a list of *`subjectaltnames`*, and optional *`bits`*. If *`bits`* is omitted a reasonable default is selected. The result is a two element list made up of a byte array with the certificate and a byte array with the private key.

`opcua datasources` *`handle ?nodeid? ?cmd?`*

Returns or modifies information on data sources (`Variable` nodes with callbacks) for the server object *`handle`*. Without optional parameters for each known data source two list elements with node identifier and callback command are added to the result. With *`nodeid`* information for the specified node is returned. With *`cmd`* the Tcl callback is changed. If *`cmd`* is given as an empty list, that callback is deleted and the variable node will return to normal (non-datasource driven) behaviour.

`opcua datetime ?seconds|...|utc` *`?value??`*

Returns either POSIX or OPC/UA timestamps as **Tcl_WideInt** values. If called without further parameters the current OPC/UA local `DateTime` is returned. If called with the single keyword `utc` the current OPC/UA *UtcTime* is returned. Otherwise, *`value`* is required and converted from POSIX to OPC/UA *UtcTime* for the keywords `seconds`, `milliseconds`, and `microseconds`, and from OPC/UA *UtcTime* to POSIX for the keywords `unixseconds`, `unixmillis`, and `unixmicros`, respectively. For the keyword `string` the *`value`* given is returned as an ISO 8601 formatted string with local time offset, for `utcstring` as an ISO 8601 formatted string in UTC. For the keyword `scan` the *`value`* given is parsed as an ISO 8601 or RFC 3339 time string and converted to an OPC/UA timestamp as **Tcl_WideInt** value.

`opcua decode` *`?handle? type bytes`*

Performs deserialization of the byte array *`bytes`* as data type *`type`*, which can be a type name or node identifier. Optionally, *`handle`* is used for type lookup. By default, if the type is `ExtensionObject`, an additional decoding step is performed to deserialize the content of that object. This can be turned off if the type is prefixed with a caret.

`opcua deftypes` *`handle nsuri defs`*

Defines custom datatypes (structures with and without optional fields, unions, and simple enumerations) in the server object *`handle`* and namespace URI *`nsuri`*. The namespace is created with the `opcua add Namespace` command and must exist before the `opcua deftypes` command is called. The parameter *`defs`* describes the structures and enumerations to be created. The command does all necessary steps to create the required nodes in the server object's address space and to store an XML bytestring describing the (de)serialization for the structures as extension objects. That XML is later to be reparsed with the `opcua gentypes` command. For details refer to section **Defining Custom Data Structures** below.

`opcua delete` *`handle`* `Node` *`nodeid ?withrefs?`*

Deletes the node with identifier *`nodeid`* on the server object *`handle`*. If *`withrefs`* is true, the references of the node are deleted, too.

`opcua delete` *`handle`* `Reference` *`srcid reftypeid targetid ?forward? ?bidir?`*

Deletes the reference described by *`srcid`*, *`reftypeid`*, and *`targetid`* on the server object *`handle`*. The boolean flag *`forward`* selects forward or inverse direction of the reference to be deleted. The boolean flag *`bidir`* requests a bidirectional reference to be deleted. The default is to delete in forward direction only.

`opcua destroy` *`handle`*

Destroys the client or server object *`handle`* and releases its resources, e.g. closes network connections, tears down the handle specific namespace, etc.

`opcua dict_init` *`name body ...`*

Helper function to create and return a dictionary for the data type *`name`* which first is primed with `opcua types empty...` and finally modified by the Tcl code in *`body`* with the same rules as in the `dict with` command. The parameters following *`body`* are assigned to the list *`args`* as in a `proc`. In contrast to `dict with`, *`body`* is executed in a call frame of its own.

`opcua disconnect` *`handle`* `?-async?`

Disconnects the client object *`handle`*. If the optional parameter `-async` is specified, the operation is carried out in asynchronous mode.

`opcua encode` *`?handle? type data`*

Performs serialization of *`data`* as data type *`type`*, which can be a type name or node identifier. Optionally, *`handle`* is used for type lookup. By default, the resulting byte array is wrapped in an `ExtensionObject`.This can be turned off if the type is prefixed with a caret.

`opcua endpoints` *`?url?`*

Queries the local OPC/UA server `opc.tcp://localhost:4840` or the server specified by the *`url`* parameter for endpoints and returns a list of URLs describing the endpoints found.

`opcua event` *`handle`* `create` *`nodeid`*

Creates an event node on the server object *`handle`*. The node identifer *`nodeid`* must be an event object type. The node identifier of the newly created event is returned. This can be used in `opcua property` commands to add information to the event, and in further `opcua event` commands to trigger it, i.e. send it out.

`opcua event` *`handle`* `oneshot` *`eventid originid`*

Sends the event node *`eventid`* on the server object *`handle`* with its origin set to node identifier *`originid`*. The event node is automatically deleted after this operation.

`opcua event` *`handle`* `trigger` *`eventid originid`*

Similar to `opcua event oneshot` but the event node is kept and can be sent again later.

`opcua fromjson` *`?handle? type json`*

This is an optional command which is available when the **open62541** library is compiled with JSON support. The JSON string *`json`* is converted to an equivalent Tcl serialization (dictionaries, lists) according to the datatype *`type`*. The *`type`* is looked up globally and optionally in the type information available for the client or server object *`handle`*. The serialized value is returned as result of the command.

`opcua genstubs` *`handle ?strip stubsts ...?`*

Generates stubs for methods in the handle specific address space derived from the client or server object *`handle`*. The address space is traversed and browse paths and node class paths are accumulated. The resulting browse paths optionally get the prefix *`strip`* stripped off from the beginning and optionally filtered using the glob patterns following the *`strip`* parameter. If *`substs`* is not empty it specifies pairwise regexps and substitutions which are applied on the browse paths for the final procedure names. For all nodes matching the node class path pattern `Object/Method` the optional `InputArguments` and `OutputArguments` child nodes are retrieved and stub procedures are written using the browse path and argument information.

`opcua gentypes` *`handle ?uri xml ...?`*

Generates custom data type mappings using information obtained from analyzing the address space derived from the client or server object *`handle`*. This feature is highly experimental and requires the tDOM package for parsing XML. It can create encoders/decoders for simple structure data types defined in the address space which perform a mapping from/to Tcl dictionaries. If the type information is to be provided directly, the optional parameters *`uri`* and *`xml`* are pairs of URI identifying a namespace and XML formatted type descriptions. The URIs are matched against the namespace array obtained from the address space of *`handle`*. For further information, see the **server_types.tcl** and **client_types.tcl** scripts in the examples directory. If this command is used, it should be invoked prior to creating method stubs, since methods may require custom data types in their arguments.

`opcua guidgen` *`?nsindex`*`|seed` *`number?`*

Generates a random GUID. If *`nsindex`* is specified, a node identifier in namespace *`nsindex`* with a random GUID is produced. If the keyword `seed` is used, the random generator is primed with current time or the optionally specified *`number`*.

`opcua history` *`handle`* `delete` *`nodeid start end`*

Deletes historic data given client *`handle`*, node identifier *`nodeid`*, and time range *`start`* and *`end`*, which must be given as OPC/UA *UtcTime* timestamps.

`opcua history` *`handle`* `insert|replace|update` *`nodeid value`*

Inserts, replaces, or updates a OPC/UA *DataValue* *value* given client *handle* and node identifier *nodeid*. The timestamps of the *DataValue* are used to select the place for the insert operation or the value to be changed, respectively.

opcua history *handle* read *nodeid start end ?numvals timestamps bounds range*?

Reads historic data given client *handle* and node identifier *nodeid*. The parameters *start* and *end* give the time range of interest as OPC/UA *UtcTime* timestamps. The *numvals* parameter limits the number of items returned. With *timestamps* the timestamps to be returned in the result are specified. With the boolean *bounds* the treatment of the upper and lower boundary of the selected set can be included or excluded. The *range* parameter must be an empty string or a valid OPC/UA index range further filtering the result set. The result is a list made up of the Tcl representation of OPC/UA *DataValues*.

opcua info *?handle?*

Returns the object type of *handle*, either client or server. If *handle* is omitted, a list of all known client and server object handles is returned.

opcua limits *handle ?name?*

Returns operation limits like **MaxNodesPerRead** or **MaxNodesPerBrowse** for the client or server *handle*. If *name* is specified that limit is returned. For both a client and server handle this involves a read operation of a variable's Value attribute below **Root/Objects/Server/ServerCapabilities/OperationLimits** when the limit isn't known yet. The value read is then cached for later re-use. If *name* is omitted a list with all limits suitable for array set is returned. In a client, limits whose values are still unknown are reported as zero. In a server the cache is filled immediately.

opcua loader *handle xml ?evar rvar tvar?*

Imports a node set from an XML string *xml* into the server *handle*. The amount of imported information highly depends on the build options of the underlying open62541 library. Node descriptions which can't be supported are ignored and internally accumulated. For analysis, that information is made available by the *evar*, *rvar*, and *tvar* result variables, which receive a list in array set form with each key being the node identifier and each value a dictionary with node information (for *evar*) including an error description, and reference information (for *rvar*), and typedef information (for *tvar*). A result is returned which is a list in array set form with each key being the method's implementation proc name and each value a list of node identifiers which will invoke the respective implementation.

opcua log *?command?*

Retrieves or sets the callback *command* for open62541 log messages. When a log message is issued, *command* is invoked with three parameters appended: the log level, e.g. info, warning, the category, e.g. network, client, and the text of the log message.

opcua mapstruct *handle ?nodeid destid members ...?*

Adds variable nodes with an internal data source mapping to the members of the structure identified by *nodeid* in the server *handle*. It allows clients to read the structure element wise even when there's no client support for deserialization of the whole structure. Alternatively, if pairs of *destid* and *members* are provided, no additional variable nodes are created but the mapping is established between the variable identified by *destid* and the structure field *members* of structure *nodeid*. If *nodeid* and following parameters are left out, the current mappings are returned as a list where each three elements are made up of destination node identifier, structure node identifier, and structure member name. **Warning:** no logic is built in to prevent from creating multiple mappings, when variable nodes are to be added. Thus, the user should ensure in this case to call the mapstruct subcommand only once per *nodeid*. Since a node only supports a single data source, an error is raised when a node already has a callback installed from the add or datasources subcommand.

opcua mbrowse *handle* {*nodeid ?dir reftype mask?*} ...

Performs a multi browse operation on the client or server object *handle* similar to opcua browse, think of a parallel version useful to reduce network latency. The nodes, browse directions and further constraints are specified in separate lists. The result is a list for each input list of arguments made up of the six elements described in *opcua browse*.

opcua methods *handle ?nodeid outtype cmd*

Returns information on methods for the server object *handle*. Without optional parameters, for each known method three list elements with node identifier, result type information, and callback command are added to the result. With *nodeid* information for the specified node is returned. With *outtype* which must be a list of dicts of serialized *Argument* structs the method's mapping of result values is modified. With *cmd*, the Tcl callback is changed. If *cmd* is given as an empty list, that callback is deleted and the method node will report an error upon call from the OPC/UA side.

Bopcua monitor *handle* configure 0 *monid ?cmd?*

Configures the monitor *monid* on the server object *handle* with the provided parameters, see opcua monitor new for further information.

opcua monitor *handle* configure *subid monid ?cmd mode interval?*

Configures the monitor *monid* in subscription *subid* on the client object *handle* with the provided parameters, see `opcua monitor new` for further information.

`opcua monitor` *handle* `destroy` *subid monid*

Destroys the monitor *monid* in subscription *subid* on the object *handle* and releases all its resources. If *handle* is a server object, *subid* must be specified as zero.

`opcua monitor` *handle* `info` *subid ?monid?*

Returns information on monitor *monid* in subscription *subid* on the *handle*. If *handle* is a server object, *subid* must be specified as zero. The result is a list of monitor type (`data` or `event`) when *handle* refers to a client object (otherwise the list element is left out), the node identifier, the callback command, the attribute, the monitor's mode (only for client handles), and the interval. If *monid* is omitted, a list of all monitor identifers registered in the subscription or in the server object is returned.

`opcua monitor` *handle* `new 0` *cmd nodeid ?attr interval?*

Creates a monitored item of the data value for the node identifier *nodeid* on the server object *handle*. The optional parameter *attr* selects the attribute of the node to be monitored (`Value` is the default). The monitoring interval <i<>`interval` must be given as number of milliseconds. The callback command parameter *cmd* isdiscussed in section **Monitor Callbacks** below. The command returns a numeric identifier of the newly created monitor.

`opcua monitor` *handle* `new` *subid type cmd nodeid ?filter attr mode interval?*

Creates a monitored item of *type* (`data` or `event`) for the node identifier *nodeid* in the subscription *subid* on the client object *handle*. The parameter *filter* must be specified for *event* monitors as a list made up of type identifiers, qualified names, and attributes, e.g. {*BaseEventType* `Message Value ...`}, where *BaseEventType* must be given as node identifier by look up per `opcua translate`. For `data` monitors, *filter* must be a list of data change trigger type, deadband type, and deadband value. If left empty, reasonable defaults are selected. The optional parameter *attr* selects the attribute of the node to be monitored (`Value` is the default for `data` monitors, `EventNotifier` for `event` monitors). The monitor mode *mode* must be one of `Disabled`, `Sampling`, and `Reporting`. The monitoring interval *interval* must be given as number of milliseconds, if omitted its value is derived from the subscription. The callback command parameter *cmd* is discussed in section **Monitor Callbacks** below. The command returns a numeric identifier of the newly created monitor.

`opcua mread` *handle cmd nodeid ...*

Similar to `opcua read` this command carries out a multi read operation of value attributes on *nodeid* and following node identifiers. If *cmd* is given as empty string, the operation is synchronous and a list of value attributes is returned. For asynchronous operation see `opcua call` and section **Aynchronous Operations**.

`opcua mreadx` *handle cmd nodeid attr index ...*

Similar to `opcua mread` this command carries out a multi read operation of attribute *attr* with *index* on *nodeid* and following parameters. Each single read is described by note identifier, attribute, and index. The *index* parameter must be an empty string or a valid OPC/UA index range. If *cmd* is given as empty string, the operation is synchronous and status codes and values for all attributes are returned as a list. For asynchronous operation see `opcua call` and section **Aynchronous Operations**.

`opcua mtranslate` *handle* {*nodeid reftype target ..*} *...*

Performs a multi translate operation on the client or server object *handle* similar to `opcua translate`, think of a parallel version useful to reduce network latency. The nodes, reference types and targets are specified in separate lists. The result is a list for each input list of arguments made up of the three elements described in `opcua translate`.

`opcua mwrite` *handle cmd nodeid type value ...*

Similar to `opcua write` this command carries out a multi write operation of value attributes on *nodeid* and following node identifiers. Each single write is described by node identifier, data type, and value. If *cmd* is given as empty string, the operation is synchronous. For asynchronous operation see `opcua call` and section **Asynchronous Operations**. In contrast to the `opcua write` command this operation transfers all values with their source timestamp set to the current time.

`opcua mwritex` *handle cmd nodeid attr index type value ...*

Similar to `opcua mwrite` this command carries out a multi write operation of attribute *attr* with *index* on *nodeid* given *type* and *value*. Each single write is described by note identifier, attribute, index, data type and value. The *index* parameter must be an empty string or a valid OPC/UA index range. If *cmd* is given as empty string, the operation is synchronous. The result is a list of status codes for each attribute given. For asynchronous operation see `opcua call` and section **Asynchronous Operations**. In contrast to the `opcua write` command this operation transfers all values with their source timestamp set to the current time.

`opcua namespace` *handle ?uri?*

Returns the namespace index for the namespace *uri* of the client or server object *handle* (or throws an

error e.g. when the namespace doesn't exist). If *uri* is omitted, a list of all known namespace indices and corresponding URIs is returned.

`opcua new ?client? ?name?`
`opcua new server port name`

Creates a new client or server object and returns its handle. The *port* parameter must be present for server objects and specifies the server's TCP port. The optional *name* is the object name (the handle). If no parameters are given to `opcua new` a client object with an automatic name is created. During that process the Tcl namespace `::opcua::`*name* is created which later is used to hold method stub procedures and other information. That namespace is tied to the life time of the client or server object. The initial access control list of a server object is empty.

`opcua onclientstate handle ?cmd?`

Returns or sets the callback for client connection state changes for the client object *handle*. The parameter *cmd* is the Tcl callback to receive connection state information; it is invoked with three added parameters: 1. the connection state as numeric OPC/UA status code, 2. the secure channel state as a string, and 3. the session state as a string. Possible secure channel states are `closed`, `hel_sent`, `hel_received`, `ack_sent`, `ack_received`, `opn_sent`, `open`, and `closing`. Possible session states are `closed`, `create_requested`, `created`, `activate_requested`, `activated`, and `closing`.

`opcua onfinalize handle ?typeid? ?cmd?`

Returns or sets the callback for node finalization events on the server object *handle*. The parameter *cmd* is the Tcl callback to handle the finalization event; it is invoked when a node in the server's address space is deleted with one additional parameter which is the node identifier of the node being deleted. *cmd* must have proper list format. If specified as an empty list, no callback on node finalization is carried out. If *typeid* is specified it must be the node identifier of an `ObjectType` or a `VariableType`. In this case, the callback is invoked when an `Object` or `Variable` node of this type is to be deleted. The callback gets two additional parameters which are the node identifiers of the deleted node and its respective type. If both, *typeid* and *cmd* parameters are omitted, a list of all finalizer callbacks is returned with the odd elements being the type node identifiers (empty element for global finalizer) and the even elements the respective callback commands.

`opcua oninitialize handle ?cmd?`

Returns or sets the callback for node initialization events on the server object *handle*. The parameter *cmd* is the Tcl callback to handle the initialization event; it is invoked when a new node is added in the server's address space with two additional parameter which are the node identifier of the new node and its node class. *Cmd* must have proper list format. If specified as an empty list, no callback on node initialization is carried out. If *typeid* is specified it must be the node identifier of an `ObjectType` or a `VariableType`. In this case, the callback is invoked when an `Object` or `Variable` node of this type is created. The callback gets two additional parameters which are the node identifiers of the new node and its respective type. If both, *typeid* and *cmd* parameters are omitted, a list of all initializer callbacks is returned with the odd elements being the type node identifiers (empty element for global initializer) and the even elements the respective callback commands.

`opcua parent handle nodeid`

Returns the parent node identifier of the given node identifier *nodeid* on the client or server object *handle*.

`opcua permissions handle user namespaceindex ?permission ...?`

Queries or modifies the user/namespace specific permissions for the server object *handle*. *user* specifies the user name from the access control list (see `opcua acl`) or an empty string for anonymous logins. *namespaceindex* is the namespace of interest in the range 0 to 31 or for changing permissions **all** to deal with all 32 namespaces at once. Namespace numbers out of range are treated like namespace zero. If no further arguments are provided, the list of permissions for that *user* and *namespaceindex* is returned. Each granted permission is indicated with a leading plus character, each revoked permission with a leading minus character, e.g. `+ACCESSLEVELMASK_READ` and `-ACCESSLEVELMASK_WRITE`. If at least one *permission* argument is specified, the same logic as for a query is used, i.e. a leading plus (minus) character grants (revokes) the respective following permission. Valid permissions are `ACCESSLEVELMASK_READ`, `ACCESSLEVELMASK_WRITE`, `ACCESSLEVELMASK_HISTORYREAD`, `ACCESSLEVELMASK_HISTORYWRITE`, `ACCESSLEVELMASK_SEMANTICCHANGE`, `ACCESSLEVELMASK_STATUSWRITE`, `ACCESSLEVELMASK_TIMESTAMPWRITE`, `EXECUTABLE`, `EXECUTABLE_ON_OBJECT`, `WRITEMASK_ACCESSLEVEL`, `WRITEMASK_ARRAYDIMENSIONS`, `WRITEMASK_BROWSENAME`, `WRITEMASK_CONTAINSNOLOOPS`, `WRITEMASK_DATATYPE`, `WRITEMASK_DESCRIPTION`, `WRITEMASK_DISPLAYNAME`, `WRITEMASK_EVENTNOTIFIER`, `WRITEMASK_EXECUTABLE`, `WRITEMASK_HISTORIZING`, `WRITEMASK_INVERSENAME`, `WRITEMASK_ISABSTRACT`, `WRITEMASK_MINIMUMSAMPLINGINTERVAL`, `WRITEMASK_NODECLASS`, `WRITEMASK_NODEID`, `WRITEMASK_SYMMETRIC`, `WRITEMASK_USERACCESSLEVEL`, `WRITEMASK_USEREXECUTABLE`, `WRITEMASK_USERWRITEMASK`, `WRITEMASK_VALUERANK`, `WRITEMASK_WRITEMASK`, `WRITEMASK_VALUEFORVARIABLETYPE`, `WRITEMASK_USERACCESSLEVEL`, `ADD_NODE`, `ADD_REFERENCE`, `DELETE_NODE`, `DELETE_REFERENCE`, `BROWSE_NODE`, `HISTORY_UPDATE_DATA`, `HISTORY_DELETE_RAW_DATA`. The permission string may be optionally prefixed with `UA_` in order to provide names as in the `open62541.h` header file. The special permission names `ACCESSLEVELMASK_ALL` and `WRITEMASK_ALL` are not reported in a query but can be used as a shortcut to turn all respective permissions on or off.

`opcua pread ...`

See section **Prepared Read And Write Operations**.

`opcua property` *`handle`* `read` *`nodeid name`*

Reads the property *`name`* from node identifier *`nodeid`* on the server object *`handle`*.

`opcua property` *`handle`* `write` *`nodeid name type value`*

Writes *`value`* of data type *`type`* to the property *`name`* of the node identifier *`nodeid`* on the server object *`handle`*.

`opcua ptree` *`handle ?nodeid? ?mask?`*

Returns information similar to `opcua tree` using the client or server object *`handle`*. The address space is traversed starting at the node identifier *`nodeid`* (the root node if omitted). The result list is made up of browse path name, node identifier, node class path, reference node identier, type node identifier, and parent node identifier. With *`mask`* the node kinds to be considered can be specified, if omitted all nodes are reported. The browse path name is a path name like notation made up of the browse names pointing to the final node as seen from the starting node. Browse names are written as qualified names, i.e. including the numeric namespace index if not in root namespace. Similarly, the node class path is a path name like notation made up of the node classes of all nodes along the path. The `opcua ptree` command is used internally by the `opcua genstubs` command in order to filter out objects and methods when creating stub Tcl commands to invoke methods on objects.

`opcua pubsub` *`handle`* `AddConnection` *`config`*

Adds a new `PubSubConnection` with the parameters from *`config`*, which must be a dictionary of type `PubSubConnectionDataType`. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `PubSubConnection` is returned.

`opcua pubsub` *`handle`* `AddDataSetFolder` *`name`*

Adds a new `DataSetFolder` object with name *`name`* in the server given the client or server object *`handle`*. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `DataSetFolder` is returned.

`opcua pubsub` *`handle`* `AddDataSetReader` *`groupId readerData`*

Adds a new `DataSetReader` object with information from *`readerData`* which must be a dictionary of type `DataSetReaderDataType` on the `ReaderGroup` with node identifier *`groupId`*. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `DataSetReader` is returned.

`opcua pubsub` *`handle`* `AddDataSetWriter` *`groupId writerData`*

Adds a new `DataSetWriter` object with information from *`writerData`* which must be a dictionary of type `DataSetWriterDataType` on the `WriterGroup` with node identifier *`groupId`*. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `DataSetWriter` is returned.

`opcua pubsub` *`handle`* `AddPublishedDataItems` *`name aliases flags vars`*

Adds a new `PublishedDataSet` with name *`name`* and the information contained in the three parameters *`aliases`* (list of `String` type), *`flags`* (list of `UInt16` type), and *`vars`* (list of `PublishedVariableDataType`). The three lists must have the same number of elements. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `PublishedDataSet` is returned.

`opcua pubsub handle AddReaderGroup` *`connId readerGroupData`*

Adds a new `ReaderGroup` object with information from *`readerGroupData`* which must be a dictionary of type `ReaderGroupDataType` on the `PubSubConnection` with node identifier *`connId`*. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `ReaderGroup` is returned.

`opcua pubsub` *`handle`* `AddWriterGroup` *`connId writerGroupData`*

Adds a new `WriterGroup` object with information from *`writerGroupData`* which must be a dictionary of type `WriterGroupDataType` on the `PubSubConnection` with node identifier *`connId`*. The operation is carried out on the server given the client or server object *`handle`*. The node identifier of the new `WriterGroup` is returned.

`opcua pubsub` *`handle`* `DeletePubSubConfiguration`

Removes all currently configured PubSub components (`PubSubConnections`, `ReaderGroups`, `WriterGroups`, `PublishedDataSets`, `DataSetReaders`, and `DataSetWriters`) in the server given the client or server object *`handle`*.

`opcua pubsub` *`handle`* `LoadPubSubConfigurationFile` *`bytes`*

This subcommand is identical to `opcua pubsubcfg load` but can be invoked from a client object *`handle`*, too.

`opcua pubsub` *`handle`* `RemoveConnection` *`connId`*

Removes the PubSubConnection with node identifier *connId*. The operation is carried out on the server given the client or server object *handle*.

`opcua pubsub` *handle* `RemoveDataSetFolder` *folderId*

Removes the DataSetFolder object with node identifier *folderId* and all contained PublishedDataSets. The operation is carried out on the server given the client or server object *handle*.

`opcua pubsub` *handle* `RemoveDataSetReader` *groupId readerId*

Removes the DataSetReader with node identifier *readerId* on the ReaderGroup with node identifier *groupId*. The operation is carried out on the server given the client or server object *handle*.

`opcua pubsub` *handle* `RemoveDataSetWriter` *groupId writerId*

Removes the DataSetWriter with node identifier *writerId* on the WriterGroup with node identifier *groupId*. The operation is carried out on the server given the client or server object *handle*.

`opcua pubsub` *handle* `RemoveGroup` *connId groupId*

Removes the ReaderGroup or WriterGroup with node identifier *groupId* on the PubSubConnection with node identifier *connId*. The operation is carried out on the server given the client or server object *handle*.

`opcua pubsub` *handle* `RemovePublishedDataSet` *pdsId*

Removes the PublishedDataSet object with node identifier *pdsId*. The operation is carried out on the server given the client or server object *handle*.

`opcua pubsubcfg` *handle* `load|save` *?bytes?*

Performs a serialization (save) or deserialization (load) of the PubSub configuration of the server object *handle*. For the load operation, *bytes* must contain a byte array of the serialization, for save a serialization is returned as result. The deserialization operation reconstructs the PubSub components in the server described by *bytes*.

`opcua pwrite ...`

See section **Prepared Read And Write Operations**.

`opcua read` *handle nodeid ?attr cmd?*

Performs a read operation on the client or server object *handle* and returns the value of attribute *attr* of the node identifier *nodeid*. If *attr* is omitted, it defaults to the Value attribute. The optional parameter *cmd* can be specified on client objects in order to carry out the read operation in asynchronous mode. See `opcua call` and section **Asynchronous Operations** for more information. If *cmd* is an empty list, on both client and server, the return value of the read operation is not the deserialized value, but a **DataValue** dictionary with additional information. The dictionary has the keys value for the value itself, valueRank with optional arrayDimensions, dataType with the node identifier of the type of the value, and optional timestamps. This mode of operation can be useful to find out if a **Variable** is a scalar or an array, or to learn the real type of a value when the related **Variable** is of an abstract type.

`opcua readjson` *handle nodeid ?attr?*

This is an optional command which is available when the **open62541** library is compiled with JSON support. Similar to `opcua read` a read operation on the client or server object *handle* is performed and the value of attribute *attr* of the node identifier *nodeid* is returned as a JSON string. If *attr* is omitted, it defaults to the Value attribute. This operation is always synchronous.

`opcua reconnect` *handle*

Reconnects the client object *handle* with the same parameters which were used upon the most recent `opcua connect` call.

`opcua reftype` *?name?*

Returns the node identifier for the reference type *name*. When *name* is omitted, a list of all reference type names is returned.

`opcua register` *handle nodelist*

Registers the node identifiers from *nodelist* in the server to which *handle* is connected. If the operation succeeds, another list of node identifiers is returned which provides aliases to the node identifiers passed to the command during the life time of the session. Using the aliases instead of the original node identifiers can improve performance of subsequent read and write operations.

`opcua request` *handle ?reqid?*

Returns information on pending asynchronous operations of the client object *handle*. If a numeric request identifier *reqid* is given, a two element list for this request is returned made up of the operation type (call, read, or write) and the callback command which receives the response. If *reqid* is omitted, a list of all known pending request identifiers is returned.

`opcua root`

Returns the node identifier of the root node.

`opcua run` *`handle ?ms?`*

Runs asynchronous operations (subscriptions, monitored items) on the client object *`handle`* for *`ms`* milliseconds. If *`ms`* is omitted, that duration defaults to zero. Normally, this operation is carried out by the Tcl event loop. Still, this command can be used to test if the client object is in the connected state.

`opcua sc2str` *`?`*`-short`*`? code`*

Translates the numeric status code *`code`* to an error message string. A single word error string such as `BadTimeout` is produced when the `-short` option is specified.

`opcua servers` *`?url?`*

Queries the local OPC/UA server `opc.tcp://localhost:4840` or the server specified by the *`url`* parameter for server information and returns a list made up of deserialized dictionaries based on the `UA_ServerOnNetwork` structure. Consult the open62541 documention for more information.

`opcua session` *`handle`* `admin|current|isadmin|list`

Queries session information from *`handle`* which must be a server handle. With parameter `current` the current session identifier during a method or data source callback or during a constructor or destructor is returned. In all other contexts, the result is an empty string. Similarly, the `isadmin` parameter reports 1 if the current session is the adminstrative (server internal) session, or 0 in all other cases. With parameter `list` a list of all known session identifiers is returned. With parameter `admin` the identifier of the adminstrative (server interal) session is returned.

`opcua start` *`handle`*

Starts the server object *`handle`*. See section **Server Object And Event Loop** below for further information.

`opcua state` *`handle`*

Reports the current state of *`handle`*. If *`handle`* refers to a client, the result is a three element list with the identical information as passed in the additional parameters to the `onclientstate` callback. Otherwise, the result is a two element list. The first element is either `stopped` or `running` indicating the server state. The second element is a dictionary with statistic counters for connections, secure channels, and sessions.

`opcua stop` *`handle`*

Stops the server object *`handle`*.

`opcua subscription` *`handle`* `configure` *`id ?interval lifetime keepalive max prio?`*

Configures the subscription *`id`* on the client object *`handle`*. See `opcua subscription new` for the optional parameters.

`opcua subscription` *`handle`* `destroy` *`id`*

Destroys the subscription *`id`* on the client object *`handle`*.

`opcua subscription` *`handle`* `info` *`?id?`*

Returns information about subscription *`id`* on the client object *`handle`* as a list of enable flag, interval, lifetime, keepalive, and maximum counters, and the priority value. If *`id`* is omitted, a list of all subscription identifiers of the client object is returned.

`opcua subscription` *`handle`* `new` *`?flag interval lifetime keepalive max prio?`*

Creates a new subscription (a container for monitored items, see `opcua monitor`) on the client object *`handle`* and returns a numeric identifier of it. The following optional parameters control properties of the subscription: *`flag`* is the initial enable state (on by default), *`interval`*, *`lifetime`*, *`keepalive`*, and *`max`* the timing and queuing parameters, and *`prio`* the subscription's priority.

`opcua subscription` *`handle`* `off` *`id`*

Disables the subscription *`id`* on the client object *`handle`*.

`opcua subscription` *`handle`* `on` *`id`*

Enables the subscription *`id`* on the client object *`handle`*.

`opcua tojson` *`?handle? type data`*

This is an optional command which is available when the **open62541** library is compiled with JSON support. The serialized (dictionaries, lists) Tcl value *`data`* is converted to an equivalent JSON string according to the datatype *`type`*. The *`type`* is looked up globally and optionally in the type information available for the client or server object *`handle`*. The JSON string is returned as result of the command.

`opcua translate` *`handle nodeid reftype target ...`*

Performs a translate operation on the client or server object *`handle`*. The operation starts at node identifier *`nodeid`* and traverses the object tree along the references *`reftype`* and browse name *`target`*. A list made up of the node identifier, namespace URI, and server index of the final target is returned as the result. References can be preceeded with an exclamation mark in order to reverse their direction. A reference may be abbreviated as slash for `HierarchicalReferences` or as dot for `Aggregates`.

`opcua tree` *`handle ?nodeid? ?mask?`*

Returns information similar to `opcua browse` using the client or server object *`handle`*. The address space is traversed starting at the node identifier *`nodeid`* (the root node if omitted). The kind of nodes to be included can be specified with *`mask`*. If *`mask`* is empty or omitted, all matching nodes are reported. The result list is made up of tree level (0-based), node identifier, browse name (qualified name), display name (locale and text), node class, reference node identifier, type node identifier, and parent node identifier.

`opcua type` *`handle nodeid ?attr?`*

Performs a read operation on the client or server object *`handle`* like `opcua read` but instead of the attribute's value returns the type name of attribute *`attr`* of the node identifier *`nodeid`*. If *`attr`* is omitted, it defaults to the `Value` attribute.

`opcua types basic|builtin|empty|info|list|map|name|nodeid` *`?handle name?`*

Returns a list of OPC/UA type names for the `basic` and `list` subcommands. Basic types are primitives (e.g. integer numbers) for which a mapping to Tcl objects is provided. The `map` subcommand returns a list of alternating node identifiers and type names suitable for `array set`. The `empty` subcommand requires *`name`* to be a known OPC/UA type name and produces and returns an empty value of this type, e.g. 0.0 for a floating point type. The `nodeid` subcommand returns the node identifier for the type name. The `name` subcommand is the reverse operation of the `nodeid` subcommand and reports the type name for node identifier in *`nodeid`*. The `info` subcommand returns detailed type information for the requested *`name`* or *`nodeid`*. For the command forms where a *`handle`* can be specified, this allows to deal with additional custom data types (see e.g. `opcua deftypes`) which where loaded into the client or server object *`handle`*. The `builtin` subcommand returns the numeric value of the data type *`name`* for primitive types of namespace zero.

`opcua unregister` *`handle nodelist`*

Unregisters the node identifiers from *`nodelist`* in the server to which *`handle`* is connected. This is the reverse of the `register` subcommand which removes the aliases to the node identifiers in *`nodelist`*.

`opcua users` *`handle`*

Returns the usernames of the currently set user/password based access control list of the server object *`handle`*. If anonymous logins are allowed, this is indicated by a username which is an empty string.

`opcua version`

Returns the major and minor version numbers of the integrated open62541 library, e.g. "1.0".

`opcua write` *`handle nodeid ?attr? type value ?cmd?`*

Performs a write operation on the client or server object *`handle`* writing *`value`* with type *`type`* into the attribute *`attr`* of the node identifier *`nodeid`*. If *`attr`* is omitted, it defaults to `Value`. The operation is performed without setting an explicit source timestamp. The optional parameter *`cmd`* can be specified on client objects in order to carry out the write operation in asynchronous mode. See `opcua call` and section **Asynchronous Operations** for more information. When the *`type`* argument is prefixed with an asterisk, the **ValueRank** of the write operation is performed as a write to an array of one or more dimensions, i.e. the *`value`* is interpreted as a list of values to be written. Likewise, when the *`type`* argument is prefixed with an exclamation mark, the *`value`* argument is treated as a scalar. For dealing with multidimensional variables (the attribute **ValueRank** indicating arrays of one or more dimensions, and the **ArrayDimensions** even explicitely stating the bounds) the special prefix for the type must be the array dimensions separated by commas and overall enclosed in parenthesis, e.g. "(3,3)i=6" or "(3,3)Int32" for an **Int32** 3x3 matrix. **Warning:** this command performs additional queries of the address space in synchronous mode (*`cmd`* argument omitted), if no prefix to the *`type`* indicating its interpretation is provided. This can cost up to two additional round-trips to the server in client mode.

`opcua writejson` *`handle nodeid ?attr? type value`*

This is an optional command which is available when the **open62541** library is compiled with JSON support. Similar to `opcua write` a write operation on the client or server object *`handle`* is performed and the attribute *`attr`* of the node identifier *`nodeid`* is written by serializing the JSON string *`value`* according to the data type *`type`*. If *`attr`* is omitted, it defaults to `Value`. This operation is always synchronous.

`opcua xcall` *`handle nodeid methodid ?type value ...?`*

This is the coroutine aware version of `opcua call`. If called with a client *`handle`* in a coroutine context it uses an asynchronous call operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

`opcua xgenstubs` *`handle ?strip substs ...?`*

> This is the coroutine aware version of `opcua genstubs`. Instead of procedure bodies using `opcua call` it writes coroutine aware procedure bodies using `opcua xcall`.

`opcua xmldump` *`handle`*

> Returns an XML string of the address space of the client or server object *`handle`*. This includes only namespaces with numeric index 2 and higher. If there are only namespace indices 0 and 1, i.e. in a fresh server without additional node sets loaded, an error is reported.

`opcua xmread` *`handle nodeid ...`*

> This is the coroutine aware version of `opcua mread`. If called with a client *`handle`* in a coroutine context it uses an asynchronous read operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

`opcua xmreadx` *`handle nodeid attr index ...`*

> This is the coroutine aware version of `opcua mreadx`. If called with a client *`handle`* in a coroutine context it uses an asynchronous read operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

`opcua xmwrite` *`handle nodeid type value ...`*

> This is the coroutine aware version of `opcua mwrite`. If called with a client *`handle`* in a coroutine context it uses an asynchronous write operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

`opcua xmwritex` *`handle nodeid attr index type value ...`*

> This is the coroutine aware version of `opcua mwritex`. If called with a client *`handle`* in a coroutine context it uses an asynchronous write operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

`opcua xread` *`handle nodeid ?attr?`*

> This is the coroutine aware version of `opcua read`. If called with a client *`handle`* in a coroutine context it uses an asynchronous read operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

`opcua xsleep` *`ms`*

> This is a coroutine aware convenience function which delays execution for *`ms`* milliseconds by using the `after` command with a callback in order to service events, if invoked in a coroutine context.

`opcua xwrite` *`handle nodeid ?attr? type value`*

> This is the coroutine aware version of `opcua write`. If called with a client *`handle`* in a coroutine context it uses an asynchronous write operation with `yield` as callback, i.e. automatically suspends the coroutine and resumes it again when the result becomes available.

OPC/UA Ensemble

The current implementation uses an ensemble and namespace `opcua`, i.e. the command `opcua info` can be alternatively written as `opcua::info`. Some more complex subcommands of the `opcua` namespace are implemented in Tcl, namely the `opcua tree` and `opcua genstubs` procedures.

Node Identifiers

Numeric node identifiers can be written as `ns=N;i=I` where `N` is the numeric namespace, and `I` the numeric identifier. Likewise, string node identifiers are written as `ns=N;s=S` with `S` being the string identifier. GUID node identifiers are written as `ns=N;g=G` where the GUID is `G` with the usual format as sequence of hexadecimal numbers and dashes. The namespace part can be left out when namespace zero is addressed. Currently, byte string node identifiers are not supported. If the format cannot be determined (e.g. since the equal sign is missing) the fallback chosen is string node identifier in namespace zero. String named namespaces are not supported.

Qualified Names

Qualified names are used for example in the `opcua browse` and `opcua translate` operations as so called browse names. These are made up of an optional numeric namespace prefix (a number followed by a colon) and a name, e.g. *`2:MyObject`*. The namespace prefix is left out if the name refers to namespace zero.

Localized Text

The data type `LocalizedText` is represented as a dict with the keys `locale` and `text`. When converting a Tcl value to a `LocalizedText` item, these keys are tried to be associated. If this operation fails, the Tcl value is used as the `text` part of the `LocalizedText` item and the `locale` part is left empty.

Supported Data Types

Currently, most of the data types of namespace zero are supported and can be mapped to/from Tcl, i.e. integral and floating point numbers, strings, GUIDs, and interal extension objects (similar to structures). For the latter, dictionaries are used in both directions, i.e. for encoding, a dictionary is searched for the respective member names, for decoding, a dictionary is created from the internal representation using the member names of the data type, see `opcua attrs default` for example. Support for custom data types is highly experimental and underdocumented (see `opcua gentypes`).

Monitor Callbacks

Monitor callbacks are invoked when a monitored item (data or event) is received. The callback parameter given in `opcua monitor new` must have proper list format and gets a single value (data) or a list of values (event) appended prior to invocation.

Data Source Callbacks

Data source callbacks are invoked when a *DataValue* is read or written to. The callback parameter given in the node creation (`opcua add Variable`) must have proper list format and gets the following parameters appended prior to invocation: the node identifier of the *DataValue*, the operation (either `read` or `write`), and the value attribute for write operations. If the read or write operation specifies a numeric range, the range description is added as the last parameter. It is expressed as a Tcl list with even number of integer elements. Each pair describes the start and end elements of a dimension. For read operations the callback must return a two element list of the data type (e.g. `String` or `Int32`) and the value itself. If the callback returns the **TCL_BREAK** return code, the value is assumed to be an array and splitted into list elements which then are converted to OPC/UA data in an OPC/UA array.

Method Callbacks

Method callbacks are invoked when a *`Method`* node is called. The callback parameter given in the node creation (`opcua add Method`) must have proper list format and gets the following parameters appended prior to invocation: the object node identifier, the method node identifier, and a list of type names derived from the method's output arguments. This list may be used in the method's implementation to form the result. The callback must return a single value which must be a list with the same number of items of the output argument's type list. Each item gets converted to the respective OPC/UA data value according to the output argument information of the `Method` node. I.e. for simple types, a list item may be a single integer e.g. for a single `UInt16`, a list of integers for an array of `UInt16`'s, a single dict for a structure type, or a list of dicts for an array of structured types. If the method invocation has to report back an error as non-zero OPC/UA status code, the method must use `return -code break`, which triggers collection of the status code from the global `errorCode` Tcl list.

Client Object And Event Loop

A client object obtained with `opcua new client` requires a running event loop only when subscriptions/monitored items are involved. Most other operations are performed synchronously (and thus blocking).

Asynchronous Operations

Asynchronous mode of operation is supported for client objects and their `opcua call`, `opcua read`, and `opcua write` subcommands. The callback parameter must have proper list format and gets a single value appended prior to invocation which carries the deserialized response message corresponding to the operation. It is made up as a nested `dict` which contains a `ResponseHeader` with the status code of the operation and depending on the operation the method call result(s) or the value of the attribute read plus additional diagnostic information. That `dict` gets an extra key named `RequestId` appended which is the integer request identifier of the operation which was returned by the command initiating the request. If the callback parameter is specified as empty list, the response is discarded, i.e. no Tcl code is evaluated and the operation appears as a one way request.

Prepared Read And Write Operations

Prepared read and write operations are modeled similar to prepared statements in databases. The operation is setup in a single preparation phase, where the node identifiers and data types (for the write direction) are specified and tied to local names. Afterpreparation the operation can be executed multiple times without the need to parse node identifiers/data types again. After execution the results can be retrieved, i.e. the data read for read operations or the write status information for write operations.

`opcua pread configure` *`phandle name ?value?`*

> Returns the configuration of the item *`name`* in *`phandle`* as dictionary of a serialized **ReadValueId**, if no *`value`* argument was provided. Otherwise changes the configuration of *`name`* according to *`value`* which must be a dictionary of a serialized **ReadValueId**.

`opcua pread delete` *`phandle`*

> Deletes the prepared read identified by <i<>phandle. This step is implicitly carried out when the OPC/UA client or server handle to which *`phandle`* is bound is deleted.

`opcua pread execute` *`phandle`*

> Performs the OPC/UA read operation described by *`phandle`* and returns an empty string as result. In error

cases an exception with the overall error result of the OPC/UA response is returned.

`opcua pread get phandle name`

Returns the last read value for item *name* of *phandle*.

`opcua pread getall phandle`

Returns a dictionary made up of the last read values of all items in *phandle*.

`opcua pread getx phandle ?name?`

Returns the last read value for item *name* of *phandle* with all available meta data in the form of a serialized **DataValue** as dictionary. If *name* is omitted, a dictionary of all items in *handle* is returned with a serialized **DataValue** for each item.

`opcua pread info handle|phandle`

Returns a list of the prepared reads of the client or server handle *handle*, or the item names of handle *phandle*.

`opcua pread new handle nodeid name ...`

Creates a new prepared read on the client or server identified by *handle* and returns a handle (*phandle*) for the prepared read. One or more items identified by *name* which will read the respective *nodeid* must be specified for the prepared read.

`opcua pread status phandle ?name?`

Returns the last status code for item *name* on *phandle*. If *name* is omitted a dictionary keyed by item names of all status codes of *phandle* is returned.

`opcua pwrite configure phandle name ?value?`

Returns the configuration of the item *name* in *phandle* as dictionary of a serialized **WriteValue**, if no *value* argument was provided. Otherwise changes the configuration of *name* according to *value* which must be a dictionary of a serialized **WriteValue**.

`opcua pwrite delete phandle`

Deletes the prepared write identified by `<i>phandle`. This step is implicitly carried out when the OPC/UA client or server handle to which *phandle* is bound is deleted.

`opcua pwrite execute phandle`

Performs the OPC/UA write operation described by *phandle* and returns an empty string as result. In error cases an exception with the overall error result of the OPC/UA response is returned.

`opcua pwrite info handle|phandle`

Returns a list of the prepared writes of the client or server handle *handle*, or the item names of handle *phandle*.

`opcua pwrite new handle nodeid type name ...`

Creates a new prepared write on the client or server identified by *handle* and returns a handle (*phandle*) for the prepared write. One or more items identified by *name* which will write the respective *nodeid* with data type *type* must be specified for the prepared write. The data type can be left as an empty string in which case it must later be provided at least once in a `opcua pwrite set` call.

`opcua pwrite set phandle name ?type? value`

Sets the value for item *name* in the prepared write *phandle*. If no data type *type* is specified, the data type of *value* is the last data type used for the item, e.g. the initial type provided in `opcua pwrite new`.

`opcua pwrite setall phandle value`

Sets the values for all items in *phandle* from the dictionary *value*.

`opcua pwrite status phandle ?name?`

Returns the last status code for item *name* on *phandle*. If *name* is omitted a dictionary keyed by item names of all status codes of *phandle* is returned.

Client Example

```
package require topcua

# create client
opcua new client C
```

```
# connect to server
opcua connect C opc.tcp://localhost:4840 user pass

# get MyNamespace
set ns [opcua namespace C MyNamespace]

# generate stub procs to methods in server
# these are created in the client specific ::opcua::C namespace
opcua genstubs C /Root/Objects/${ns}:MyObject/${ns}:

# list all procs in client specific namespace
puts stderr [info procs ::opcua::C::*]

# call stubs
puts stderr [::opcua::C::Reverse esreveR]
puts stderr [::opcua::C::WordSplit "word\n\nsplit"]

# read a variable
puts stderr [opcua read C "ns=${ns};ItsTclTime"]

# monitor callback proc
proc monitor {data} {
    puts stderr "Monitor: $data"
}

# make a subscription
set sub [opcua subscription C new 1 1000.0]

# make a monitor
set mon [opcua monitor C new $sub data monitor "ns=${ns};ItsTclTime"]
puts stderr "Subscription: $sub"
puts stderr "Monitor: $mon"

# handle monitors for a few seconds
set done 0
after 10000 {set done 1}
vwait done

# delete monitor and subscription
opcua monitor C destroy $sub $mon
opcua subscription C destroy $sub

# shut down the server using a method call
::opcua::C::Exit

# destroy the client
opcua destroy C
```

Server Object And Event Loop

A server object obtained with `opcua new server` requires a running event loop as long as it is in running state (started with `opcua start`). Depending on the support of the underlying open62541 library, the opcua server's network handler re-dispatches itself using a Tcl timer callback whose interval is controlled by the protocol timers of the OPC/UA stack implementation, or it spins up a dedicated thread which deals with the network traffic. In the latter case, a running event loop is required, too, for processing method and datasource callbacks.

Server Example

```
package require topcua

# create server
opcua new server 4840 S

# setup access control
opcua acl S user pass

# implementations of methods etc.
namespace eval ::opcua::S {
    # method callback
    proc _reverse {obj meth string} {
        return [string reverse $string]
    }
    # method callback
    proc _wordsplit {obj meth string} {
        set w [regexp -all -inline {\S+} $string]
        # return code break makes into an array result
        return -code break $w
    }
    # method callback
    proc _exit {obj meth} {
        after 1000 [namespace current]::_real_exit
        return {}
```

```
        }
        # helper proc
        proc _real_exit {} {
            catch {
                ::opcua::stop S
                ::opcua::destroy S
            }
            exit 0
        }
        # data source callback
        proc _its_tcl_time {node op {value {}}} {
            if {$op eq "read"} {
                return [list String [clock format [clock seconds]]]
            }
            return {}
        }
    }

    # create our OPC/UA namespace
    set ns [opcua add S Namespace MyNamespace]

    # get Objects folder
    set OF [lindex [opcua translate S [opcua root] / Objects] 0]

    # create an object in our namespace in Objects folder
    set obj [opcua add S Object "ns=$ns;s=MyObject" $OF Organizes \
        "$ns:MyObject"]

    # create methods on object
    set meth [opcua add S Method "ns=$ns;s=Reverse" \
                  $obj HasComponent \
                  {String !out} "$ns:Reverse" {String !in} \
                  ::opcua::S::_reverse]
    set meth [opcua add S Method "ns=$ns;s=WordSplit" \
                  $obj HasComponent \
                  {String !out} "$ns:WordSplit" {String !in} \
                  ::opcua::S::_wordsplit]
    set meth [opcua add S Method "ns=$ns;s=Exit" \
                  $obj HasComponent \
                  {} "$ns:Exit" {} \
                  ::opcua::S::_exit]

    # create a variable in our namespace in Objects folder
    set var [opcua add S Variable "ns=$ns;s=ItsTclTime" \
                 $OF Organizes \
                 "$ns:ItsTclTime" {} {} \
                 ::opcua::S::_its_tcl_time]

    # dump methods
    puts stderr [opcua methods S]

    # generate stubs to methods in server
    # these are created in the server specific ::opcua::S namespace
    opcua genstubs S /Root/Objects/${ns}:MyObject/${ns}:

    # list all procs in server specific namespace
    puts stderr [info procs ::opcua::S::*]

    # call stubs directly on server
    puts stderr [::opcua::S::Reverse esreveR]
    puts stderr [::opcua::S::WordSplit "word\n\nsplit"]

    # read our variable
    puts stderr [opcua read S $var]

    # start server
    opcua start S

    # enter event loop
    vwait forever
```

Defining Custom Data Structures

```
    package require topcua

    # create server
    opcua new server 4840 S

    # create our namespace
    set NS http://www.androwish.org/TestNS/
    set nsidx [opcua add S Namespace $NS]

    # create structs etc., field names prefixed with '*' are arrays
```

```
# CAUTION: no comments allowed in the definition list
opcua deftypes S $NS {
    typedef WORD UInt16
    struct KVPair {
        String name
        String value
    }
    struct RGB {
        WORD red
        WORD green
        WORD blue
    }
    struct NamedColor {
        String name
        RGB color
    }
    struct WithArray {
        String name
        String *values
    }
    enum SimpleEnum { Red Green Blue }
    union Various {
        KVPair pair
        RGB color
        NamedColor ncolor
    }
    optstruct OptColor {
        mandatory String name
        optional RGB color
    }
}

# import type defs
opcua gentypes S

# make some variables using the structs from above
set OF [lindex [opcua translate S [opcua root] / Objects] 0]
foreach {name type} {
    X1 KVPair
    X2 RGB
    X3 NamedColor
    X4 WithArray
} {
    set att [opcua attrs default VariableAttributes]
    dict set att dataType [opcua types nodeid S $type]
    dict set att value [list $type [opcua types empty S $type]]
    opcua add S Variable "ns=${nsidx};s=$name" $OF Organizes \
        "${nsidx}:$name" {} $att
}

opcua write S "${nsidx}:X4" Value WithArray {
    name {A B C D E}
    values {A B C D E}
}

# start server
opcua start S

# enter event loop
vwait forever
```

topcua::filesystem

opcua::filesystem

Name

opcua::filesystem - OPC/UA filesystem for **topcua**

Synopsis

```
package require topcua::filesystem
opcua::filesystem handle foldername rootdir
opcua::fsdestroy handle
opcua::fsrescan handle
```

Description

These commands provide several operations to manage **FileType** and **FileDirectoryType** objects in an **opcua** based OPC/UA server. The parameter *handle* in all commands must refer to a server handle. *Foldername* is the browse and display name for the toplevel node (the root) of the OPC/UA file system and is located in the **/Root/Objects** folder. *Rootdir* is the native root directory to be mapped.

The `opcua::filesystem` command performs the mount operation and creates the required OPC/UA nodes to map to native files and directories below *rootdir*.

The `opcua::fsdestroy` command performs the unmount operation and destroys all related OPC/UA nodes and closes open files.

The `opcua::fsrescan` command synchronizes the OPC/UA address space below the node corresponding to *foldername* to the contents of the native *rootdir*.

Up to 256 open native files are managed. By using session identifiers tied to open files internally, the maximum number of open files per session is limited to 32. When files are opened and closed, an automatic cleanup for orphanded session identifiers removes left over open files. Since open files are tied to sessions a natural isolation between open files and foreign sessions is achieved.

Native file names containing forward or backward slashes, colons, vertical bars, or tilde characters are not mapped into the OPC/UA address space.

Methods implemented for FileType

`UInt32 Open (Byte` *mode)*

> Returns an integer handle for the open file. *Mode* is a bitmask with bits 0=read, 1=write, 2=truncate, 4=append.

`void Close (UInt32` *handle*`)`

> *Handle* is an integer handle of an open file which is to be closed.

`ByteString Read (UInt32` *handle*`, Int32` *length*`)`

> *Handle* is an integer handle of an open file from which up to *length* bytes are to be read.

`void Write (UInt32` *handle*`, ByteString` *data*`)`

> *Handle* is an integer handle of an open file to which *data* is to be written.

`UInt64 GetPosition (UInt32` *handle*`)`

> *Handle* is an integer handle of an open file for which the current file position is to be returned.

`void SetPosition (UInt32` *handle*`, UInt64` *pos*`)`

> *Handle* is an integer handle of an open file whose file position is to be set to *pos*.

Properties implemented for FileType

`UInt64` *Size*

> The actual size of the file in bytes.

`UInt16` *OpenCount*

> How many open file handles exist for the file.

`Boolean` *`Writable`*

True when the file is writable.

`Boolean` *`UserWritable`*

True when the file is writable by the user.

Methods implemented for FileDirectoryType

`NodeId CreateDirectory (String` *`name`*`)`

`Name` is the name of the new directory to be created.

`UInt32 CreateFile (String` *`name`*`, Boolean` *`keepOpen`*`)`

`Name` is the name of the new file to be created. If *`keepOpen`* is true, the file is opened for reading and writing and its handle returned.

`void Delete (NodeId` *`toDelete`*`)`

`ToDelete` is the node identifier of the file or directory to be deleted.

`NodeId MoveOrCopy (NodeId` *`src`*`, NodeId` *`dst`*`, Boolean` *`copy`*`, String` *`name`*`)`

`Src` is the source file or directory, *`dst`* the destination directory. If *`copy`* is true, a copy operation is to be carried out. *`Name`* is the name of the new file or directory. Leave it empty for move operations. In all cases the node identifier of the resulting object is returned.

Non-standard FileDirectory methods available on toplevel node

`void CloseAll (void)`

Closes all open file handles unconditionally.

`void Rescan (void)`

Performs identical operation of the `opcua::fsrescan` command.

Error reporting

Exceptions of method calls are reported as non-zero status codes, e.g. **BADINVALIDARGUMENT** (0x80ab0000).

topcua::sqlmodel

opcua::sqlmodel

Name

opcua::sqlmodel - SQLite based address space models for **topcua**

Synopsis

```
package require topcua::sqlmodel
opcua::sqlmodel::export handle ...
opcua::sqlmodel::import handle ...
opcua::sqlmodel::ns0full handle
opcua::sqlmodel::mkspecsdb dbname ...
opcua::sqlmodel::loadspec handle name ...
opcua::sqlmodel::listspecs ?dbname?
opcua::sqlmodel::getunece ?dbname?
```

Description

These commands provide several operations to deal with an OPC/UA address space using SQLite as a storage backend. Additional commands allow for loading companion specs in compressed XML format from a central per-user database.

`opcua::sqlmodel::export` *handle* `-file` *name*

> Exports the OPC/UA address space of *handle* to the SQLite database *name*. The database file is created, temporarily opened, written, and finally closed.

`opcua::sqlmodel::export` *handle* `-db` *db* `?-schema` *schema*`?`

> Exports the OPC/UA address space of *handle* to the SQLite database handle *db* which must refer to an open, writable SQLite database. Optional *schema* is the table name prefix in case an attached database below *db* shall be written.

`opcua::sqlmodel::export` *handle* `-data` *varname*

> Exports the OPC/UA address space of *handle* to variable *varname* as a serialization of an SQLite database.

`opcua::sqlmodel::export` *handle* `-chan` *chan*

> Exports the OPC/UA address space of *handle* to an open and writable channel *chan* which is written with a serialization of an SQLite database.

`opcua::sqlmodel::import` *handle* `-file` *name*

> Imports into the OPC/UA address space of *handle* from the SQLite database *name*. Only non-existing nodes, references, and data type information is imported. The database file is temporarily opened, read, and finally closed.

`opcua::sqlmodel::import` *handle* `-db` *db* `?-schema` *schema*`?`

> Imports into the OPC/UA address space of *handle* from the SQLite database handle *db* which must refer to an opened, readable SQLite database. Optional *schema* is the table name prefix in case an attached database below *db* shall be used for the import. Only non-existing nodes, references, and datatype information is imported.

`opcua::sqlmodel::import` *handle* `-data` *value*

> Imports into the OPC/UA address space of *handle* from a serialized SQLite database in *value*.

`opcua::sqlmodel::import` *handle* `-chan` *chan*

> Imports into the OPC/UA address space of *handle* from a serialized SQLite database which is read form the open and readable channel *chan*.

`opcua::sqlmodel::ns0full` *handle*

> Loads full namespace zero into *handle* from a local database file **ns0.db.gz** which must be located in the directory where this package has been installed.

`opcua::sqlmodel::mkspecsdb` *dbname* `?`*tag*`?`

> Tries to create a database of OPC/UA companion specs into *dbname*. The XML nodeset data is obtained from a ZIP which is downloaded from github. *tag* is the git tag, **latest** is the default tag. The database is

space optimized and has the tables **Models**, **Requires**, and **UNECE** which provide meta information of the companion specs and the corresponding XML nodeset data in gzip format.

`opcua::sqlmodel::loadspec` *handle name ?dbname?*

Looks up the companion spec *name* in the database *dbname* or if *dbname* is omitted in **$::env(HOME)/uaspecs.db** and loads the corresponding XML into *handle* which must refer to a server object. *name* may be given as URL or as short name, e.g. **DI**. The loading process inspects both the current server address space and the database and tries to resolve required companion specs automatically.

`opcua::sqlmodel::listspecs ?`*dbname*`?`

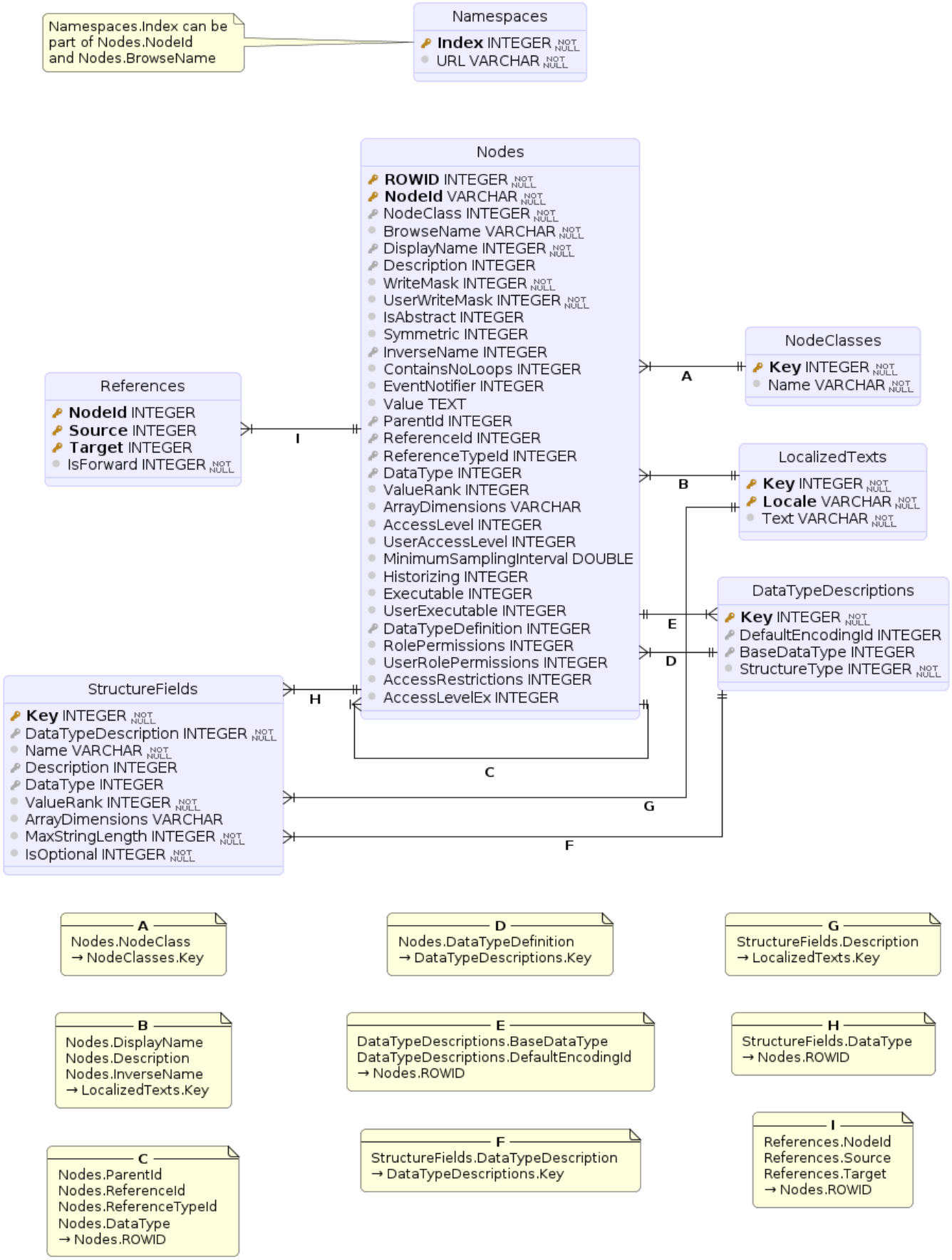
List companion specs in the database *dbname* or if *dbname* is omitted in **$::env(HOME)/uaspecs.db**. The result is a list of alternating short names and URLs.

`opcua::sqlmodel::getunece ?`*dbname*`?`

Return UNECE information from database *dbname* or if *dbname* is omitted from **$::env(HOME)/uaspecs.db** as list of alternating UNECE code, unit identifier, display name, and description.

Database schema for OPC/UA address space

In contrast to XML models the SQL data model is a full description of the address space, i.e. always includes all namespaces starting from zero. The import process initially examines the current state of the address space and adds only those nodes from the SQL model which are not existing yet.




```
CREATE TABLE Nodes(
    NodeId VARCHAR PRIMARY KEY NOT NULL,
    NodeClass INTEGER NOT NULL DEFAULT 0,   -- ref to NodeClasses
    BrowseName VARCHAR NOT NULL,
    DisplayName INTEGER NOT NULL DEFAULT 0, -- ref to LocalizedTexts
    Description INTEGER,                    -- ref to LocalizedTexts
    WriteMask INTEGER NOT NULL,
    UserWriteMask INTEGER NOT NULL,
    IsAbstract INTEGER,
```

```
    Symmetric INTEGER,
    InverseName INTEGER,                    -- ref to LocalizedTexts
    ContainsNoLoops INTEGER,
    EventNotifier INTEGER,
    Value TEXT,
    ParentId INTEGER,                       -- ROWID of Nodes
    ReferenceId INTEGER,                    -- ROWID of Nodes
    ReferenceTypeId INTEGER,                -- ROWID of Nodes
    DataType INTEGER,                       -- ROWID of Nodes
    ValueRank INTEGER,
    ArrayDimensions VARCHAR,
    AccessLevel INTEGER,
    UserAccessLevel INTEGER,
    MinimumSamplingInterval DOUBLE,
    Historizing INTEGER,
    Executable INTEGER,
    UserExecutable INTEGER,
    DataTypeDefinition INTEGER,             -- ref to DataTypeDescriptions
    RolePermissions INTEGER,
    UserRolePermissions INTEGER,
    AccessRestrictions INTEGER,
    AccessLevelEx INTEGER
);

CREATE TABLE LocalizedTexts(
    "Key" INTEGER NOT NULL,
    Locale VARCHAR NOT NULL DEFAULT "",
    Text VARCHAR NOT NULL,
    PRIMARY KEY("Key", Locale)
);

CREATE TABLE NodeClasses(
    "Key" INTEGER PRIMARY KEY NOT NULL,
    Name VARCHAR NOT NULL
);

CREATE TABLE DataTypeDescriptions(
    "Key" INTEGER PRIMARY KEY NOT NULL,
    DefaultEncodingId INTEGER,              -- ref to Nodes
    BaseDataType INTEGER,                   -- ref to Nodes
    StructureType INTEGER NOT NULL
);

CREATE TABLE StructureFields(
    "Key" INTEGER PRIMARY KEY NOT NULL,
    DataTypeDescription INTEGER NOT NULL,   -- ref to DataTypeDescriptions
    Name VARCHAR NOT NULL,
    Description INTEGER,                    -- ref to LocalizedTexts
    DataType INTEGER,                       -- ref to Nodes
    ValueRank INTEGER NOT NULL,
    ArrayDimensions VARCHAR,
    MaxStringLength INTEGER NOT NULL DEFAULT 0,
    IsOptional INTEGER NOT NULL DEFAULT 0
);

CREATE UNIQUE INDEX StructureFieldsIndex1
    ON StructureFields(DataTypeDescription, Name);

CREATE TABLE "References"(
    NodeId INTEGER,                         -- ref to Nodes
    Source INTEGER,                         -- ref to Nodes
    Target INTEGER,                         -- ref to Nodes
    IsForward INTEGER NOT NULL DEFAULT 1,
    PRIMARY KEY(NodeId, Source, Target, IsForward)
);

CREATE TABLE Namespaces(
    "Index" INTEGER PRIMARY KEY NOT NULL,
    URL VARCHAR NOT NULL
);
```

Database schema for Companion Specs etc.

```
CREATE TABLE Models(
    Model VARCHAR NOT NULL,                 -- URI
    Name VARCHAR UNIQUE NOT NULL,           -- short name
    Version VARCHAR DEFAULT '',
    PublicationDate VARCHAR DEFAULT '',
    XML BLOB NOT NULL,                      -- gzip compressed
    PRIMARY KEY(Model)
);

CREATE TABLE Requires(
```

```
     Model VARCHAR NOT NULL,                -- URI
     RequiredModel VARCHAR NOT NULL,
     RequiredVersion VARCHAR DEFAULT '',
     RequiredPublicationDate VARCHAR DEFAULT '',
     PRIMARY KEY(Model, RequiredModel)
);

CREATE TABLE UNECE(
     UNECECode VARCHAR NOT NULL,
     UnitId VARCHAR NOT NULL,
     DisplayName VARCHAR NOT NULL,
     Description VARCHAR NOT NULL,
     PRIMARY KEY(UNECEcode)
);
```

undroidwish

undroidwish

AndroWish sans the borg,
a project just for pun.

Experimental. This is a single-file Tcl/Tk binary for Windows (32 bit, optional 64 bit) and Linux using parts of the AndroWish source tree, in particular the ZIP virtual file system and the SDL/AGG/freetype based X11 emulation for rendering. So far it is a proof of concept which eventually can be extended to run on another fruity smartphone platform. It is built by executing platform dependent shell scripts which are available for Windows, Linux, and other platforms. Ready-made binaries for 32 and 64 bit Windows and Intel Linux are listed on the Downloads page. It is possible to build undroidwish on Debian platforms with ARM processors like the Raspberry Pi or the Beaglebone.

Warning! undroidwish.exe is a Windows 32 bit binary which like other nicely playing portable apps does not write to the registry or otherwise modifies the system. But running it on your Windows PC is at your own risk. It is believed to be a CAREFUL (Click And Run Executable For Unplanned Leisure) thing. Although in the first place it might look like Tk in an X11 server, it provides all the benefits of the underlying AGG/SDL2/freetype based X11 emulation, i.e. anti-aliased rendering of lines, circles, and fonts. It even allows to smoothly zoom the Tk root window by using the mouse wheel combined with the control key.

Wayland. Another build script is provided which allows building undroidwish with the SDL2 Wayland video driver. This is partially tested on the GNOME based Fedora 26-29 Workstation, Debian 9 "Stretch", and CentOS 7.5. As of 2018-02-16 this variant is built with the KMSDRM SDL2 video driver enabled, which allows to run from a console without requiring any display manager infrastructure, provided that the Linux system has decent graphics hardware allowing for kernel mode setting and direct render mode.

FreeBSD and OpenBSD. These are very similar to the Linux version (including almost all extensions) but only partially tested on FreeBSD-11 on x86 processors and OpenBSD-6.2 on amd64 processors.

OpenIndiana Hipster (based on illumos, based on SunOS 5.11). As for FreeBSD with many extensions but only partially tested in a 32 bit enviroment.

MacOS. Alpha versions are available since 2017-09-01, but are only partially tested on MacOS 10.11 (El Capitan) and 10.13 (High Sierra).

Haiku. Partial support for the Haiku operating system is now available thanks to SDL2's video driver architecture. This is still highly experimental.

```
    There are Tk ports
    one of them is undroidwish
    which runs on Haiku
```

Raspberry Pi. A Raspberry specific video driver called RPI is available in SDL2 which provides a similar feature set as the KMSDRM driver, i.e. allows to run undroidwish in frame buffer mode. When built for/on the Raspberry this driver is turned on by default, provided a recent Debian 9 (Raspbian) is used as build environment.

jsmpeg Video Driver. This is a special video driver which is described further in jsmpeg SDL Video Driver. It allows to direct the undroidwish display to a page in a modern web browswer, e.g. Firefox, Safari, or Chrome. The feature is available starting with the "Eppur si muove (2019-06-22)" release in most Linux, Windows, and MacOSX variants of undroidwish.

All undroidwish variants have many of the advanced Tcl/Tk extensions from Batteries Included built in: `tkpath`, `tktreectrl`, `tkimg`, and `Canvas3D` (which requires the display driver to support OpenGL 2.x or better). Tcl-only extensions (without machine specific libraries) like `tcllib`, `tksqlite`, and `bwidgets` are included, too.

Some SDL specific command line options described in Beyond AndroWish can be used to control the size of the Tk root window or its resizability. Other SDL specific things can be controlled at runtime using the sdltk command.

In order to start built in scripts directly (which were baked into the ZIP file system), the script to be executed must be specified on the command line with its path within the embedded ZIP file system. Here are some examples.

The widget demo

```
    undroidwish.exe builtin:sdl2tk8.6/demos/widget
```

TkSQLite, a graphical frontend to SQLite databases

```
    undroidwish.exe builtin:tksqlite0.5.13/tksqlite.tcl
```

The PostScript tiger, a tkpath demo

```
undroidwish.exe builtin:tkpath0.3.3/demos/tiger.tcl
```

Canvas3D demo, multiple threads

```
undroidwish.exe builtin:Canvas3d1.2.4/demo/threads.tcl
```

Canvas3D demo of VR rendering

```
undroidwish.exe builtin:Canvas3d1.2.4/demo/vr_chick.tcl
```

Some shortcuts are provided as shown in the table below.

Script URL	Description
builtin:widget	The widget demo
builtin:tksqlite	Graphical frontend to SQLite databases
builtin:imgdemo	Supported image formats
builtin:tkpdemo	TkPath demo
builtin:3ddemo	Canvas3D demo
builtin:tkcon	Tk console
builtin:treectrl	Tree control widget demo
builtin:tktable	Table widget demo
builtin:bugz	See Tk_Bugz in Tcl'ers Wiki, playable with a game pad
builtin:tkchat	TkChat instant messaging application
builtin:zint	Demo for ZINT barcode generator
builtin:sdx	SDX utility
builtin:dungfork	Read-only /etc browser demo using tcl-augeas
builtin:vncviewer	Simple VNC viewer using tkvnc
builtin:notebook	Will Duquette's Notebook App
builtin:tkmc	Simple clone of Midnight Commander from Tcl'ers wiki
builtin:zinc-widget	Tkzinc demo
builtin:tkinspect	Tool to inspect other running Tk applications
builtin:stardom	Small XML browser/editor
builtin:helpviewer	tkhtml based help file viewer
builtin:mpksc	mpexpr based calculator
builtin:tixtour	Demo of tix widgets
builtin:tixwidgets	Another demo of tix widgets
builtin:TDK/checker	vanillawish only: Tcl Dev Kit checker
builtin:TDK/compiler	vanillawish only: Tcl Dev Kit compiler
builtin:TDK/debugger	vanillawish only: Tcl Dev Kit debugger
builtin:TDK/inspector	vanillawish only: Tcl Dev Kit inspector
builtin:TDK/tape	vanillawish only: Tcl Dev Kit tape
builtin:TDK/tclapp	vanillawish only: Tcl Dev Kit tclapp
builtin:TDK/tclsvc	vanillawish only: Tcl Dev Kit tclsvc
builtin:TDK/vfse	vanillawish only: Tcl Dev Kit vfse
builtin:tclline	vanillawish only:pure Tcl readline
builtin:ased	vanillawish only (linux/windows): ASED Tcl/Tk IDE
builtin:vtcl	vanillawish only (linux/windows): Visual Tcl
builtin:cwsh	vanillatclsh only: Curses like Tk wish
builtin:mktclsh	vanillawish only (windows): extract a vanillatclsh
builtin:critcl	vanillawish (linux): stripped down version of critcl
builtin:ckua	vanillatclsh only: text mode OPC/UA server/browser
builtin:tdbcsh	vanillatclsh only: text mode TDBC frontend
builtin:LUCK	vanillawish: LUCK build system
builtin:lucktui	vanillatclsh only: text mode LUCK build system/frontend
builtin:tcled	vanillatclsh only: text mode editor
builtin:TSB	vanillawish: Taygete Scrap Book, see Tcl'ers wiki
builtin:fuse	Export the ZIP content of the binary per FUSE mount

usbserial command

usbserial command

Name

usbserial - transfer data over USB-serial converters

Synopsis

```
package require Usbserial
usbserial ?devicename?
```

Description

This command is used to transfer data over supported USB-serial converters (FTDI, CDC, Prolific, etc.), see this reference. When no further argument is given to the `usbserial` command, a list of supported USB device names in the form of zero or more `/dev/bus/usb/MMM/NNN` device special file names is returned. When the USB device name of a supported USB-serial converter is given as argument, `usbserial` opens that USB device and returns a Tcl channel handle for it. This handle may be used with `fconfigure`, `gets`, `read`, `puts`, and `close`. The options `-mode`, `-ttycontrol`, and `-ttystatus` to `fconfigure` are supported by the channel. However, support for getting and/or setting control lines varies between different USB-serial converter chips. Note, that similar to a normal POSIX tty device an USB device name can be opened more than once simultaneously.

List of supported devices

Vendor ID	Product ID	Remarks
0x10c4	0xea60	CP2102
0x10c4	0xea70	CP2105
0x10c4	0xea71	CP2108
0x10c4	0xea80	CP2110
0x067b	0x2303	Prolific PL2303
0x0403	0x0601	FTDI FT232R
0x0403	0x6015	FTDI FT231X
0x2341	0x0001	Arduino UNO
0x2341	0x0010	Arduino Mega 2560
0x2341	0x003b	Arduino Serial Adapter
0x2341	0x003f	Arduino Mega ADK
0x2341	0x0042	Arduino Mega 2560 R3
0x2341	0x0043	Arduino UNO R3
0x2341	0x0044	Arduino Mega ADK R3
0x2341	0x8036	Arduino Leonardo
0x16c0	0x0483	TeensyDuino
0x03eb	0x2044	ATMEL LUFA CDC Demo Application
0x1eaf	0x0004	Leaflabs Maple
0x1a86	0x7523	CH 34x
0x1a86	0x5523	CH 34x
0x4348	0x5523	CH 34x

uvc command

Name

uvc - Interface to UVC cameras using `libuvc`

Synopsis

```
package require tcluvc
uvc option ?arg ...?
```

Description

This command provides several operations to interface UVC USB cameras using the infrastructure provided by `libuvc` and `libusb` which is available on common Linux, FreeBSD, and MacOSX platforms and sometimes found working on Android devices. *option* indicates what to carry out. Any unique abbreviation for *option* is acceptable. The valid options are:

`uvc close` *devid*

> Closes the device identified by *devid* which has been opened before using `uvc open`.

`uvc convmode` *devid* ?*flag*?

> Reports or modifies the conversion mode for frames acquired from the opened device identified by *devid*. Conversion mode 1 (on/true) performs frame format/color space conversions in the special UVC thread which controls the USB transfers, mode 0 (off/false) does this instead in the normal Tcl event loop. The default mode is 1.

`uvc counters` *devid*

> Reports a three element list of statistic counters on the device identified by *devid*. The first element is the number of video frames received, the second the number of video frames processed with `uvc image`, and the third the number of video frames dropped.

`uvc devices`

> Returns device information which can be used for `uvc open` as a list. Each device adds three elements to the list: the first element is the device name as a colon separated string with two or three fields being vendor ID (hexadecimal, 0x prefix is optional), product ID (hexadecimal, 0x prefix is optional), and bus/device numbers separated by a dot; the second and third list elements are the vendor name, and the product name. To open the device, its name (the colon separated string) must be used, the other two items are available for presentation purposes. If `udev` support is available (Linux specific), this list is refreshed on plug and unplug of devices. Otherwise, the list is a snapshot of suitable devices currently connected.

`uvc format` *devid* ?*index fps*?

> Returns or changes the frame format of the device identified by *devid*. The optional parameter *index* is an integer number giving the index of the frame format returned in `uvc listformats`. The optional parameter *fps* is the frame rate. If omitted, the currently active index and frame rate are returned. Changing the frame format and rate is only possible if the device is not capturing images.

`uvc greyshift` *devid* ?*shift*?

> Returns or sets the bit shift to be applied on grey images with a bit depth higher than 8 which are captured from device *devid*. The default value is 4, which is suitable for greyscale cameras with 12 bit resolution. The shift is not applied when the `image` subcommand retrieves raw byte array data.

`uvc image` *devid* ?*photoImage*?

> Copies the most recent captured image of the device *devid* into the photo image identified by *photoImage* and returns non-zero on success or zero if no data transfer has taken place. If *photoImage* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel, and the last the image's RGB values with 3 bytes per pixel in red, green, blue order as a byte array. In this case an error is indicated by throwing an exception.

`uvc info` ?*devid*?

> Returns information on open devices. If *devid* is specified, a list of two elements is returned, the first being the device nameand the second the image callback command for that device, i.e. the same arguments which were used on `uvc open`. If *devid* is omitted, a list of *devid*s, i.e. all currently opened devices is returned.

`uvc listen ?callback?`

Retrieves or sets the callback command called on plug and unplug of devices. When a device is plugged or unplugged that callback is invoked with two additional arguments: the type of event (`add` or `remove`) and the device name (see `uvc devices` for the naming convention) which was added or removed. Only usable if `udev` support is available.

`uvc listformats devid`

Returns a dictionary keyed by a format index as integer with the values being another dictionary with information about the frame size and rate of the respective frame format. The returned indices can be used in in `uvc format` to switch to another frame size and/or to change the frame rate.

`uvc mbcopy bytearray1 bytearray2 mask`

Copies the content of RGB byte array *bytearray2* into the byte array *bytearray1* using an RGB *mask*. Both byte arrays must have identical length which must be a multiple of 3 (for RGB). The main purpose of this command is to combine images from two cameras into an anaglyph 3D, where (for a red-cyan anaglyph) the left camera image uses mask 0xFF0000 (red component) and the right camera image uses mask 0x00FFFF (green and blue components).

`uvc mcopy photo1 photo2 mask`

Copies the content of the photo image *photo2* into the photo image *photo1* using an ARGB *mask*. Both photo images must have identical width, height, and depth. The main purpose of this command is to combine images from two cameras into an anaglyph 3D, where (for a red-cyan anaglyph) the left camera image uses mask 0x00FF0000 (red component) and the right camera image uses mask 0x0000FFFF (green and blue components).

`uvc mirror devid ?x y?`

Retrieves or sets flags to mirror captured images along the X or Y axis. Parameters *x* and *y* if specified must be boolean values.

`uvc open devname callback`

Opens the device with device name *devname* and establishes *callback* as command to be invoked on captured images and finally returns a *devid*, i.e. a handle to further deal with the device. An additional parameter is appended when *callback* is invoked: the *devid* of the device. For the format of *devname* see the description of `uvc devices`.

`uvc orientation devid ?degrees?`

Retrieves or sets the orientation of captured images regarding image rotation. *degrees* if specified must be an integer number.

`uvc parameters devid ?key value ...?`

Returns or changes device parameters for the device identified by *devid* given as key-value pairs, e.g. `brightness 100` will change the brightness setting of captured images to the device dependent value 100. The command returns the current device parameters (after the potential change, when keys and values were given) as a key-value list which can be processed with `array set` or `dict get`.

`uvc record devid frame width height bpp bytearray`

Transcodes the frame described by *width*, *height*, *bpp*, and *bytearray* to JPEG and writes the result to the recording file or stream. The recording must have been started with the `-user` option. An integer number is returned as result: 1 indicates successful write, 0 no write due to frame rate constraints, and -1 an error during the write.

`uvc record devid pause`

Pauses recording to a file or stream.

`uvc record devid resume`

Continues recording to a file or stream.

`uvc record devid start options ...`

Starts recording to a file or stream. *options* control the data format, frames per second, and output channel. The option `-fps` specifies the approximate rate in frames per second as a floating point number. The option `-chan` specifies the channel to which the frames are written. This channel is detached from the Tcl interpreter and controlled solely by the `uvc record` command. The `-boundary` option specifies a MIME multipart boundary string and selects the MIME type `multipart/x-mixed-replace` suitable for streaming to a web browser. The content type delivered to the browser is `image/jpeg`. If the `-boundary` option is omitted, the output format is raw AVI and requires the channel to be seekable. The option `-mjpeg` forces the recorded data to JPEG format, i.e. a transcoding to JPEG will be performed in software, if the device doesn't already deliver a JPEG stream. The option `-user` turns off automatic frame write operations to the recording file or stream when a frame is delivered from the device. Instead, `uvc record devid frame` must be invoked in the callback function. The `-user` option implies `-mjpeg`.

`uvc record` *`devid`* `state`

Returns the current recording state as `stop`, `recording`, `pause`, or `error`. The state `error` indicates a write error on the file or stream. In this case no further frames will be written.

`uvc record` *`devid`* `stop`

Finishes recording to a file or stream and closes the underlying channel.

`uvc start` *`devid`*

Starts capturing images of the device identified by *`devid`*. When an image is ready, the callback command set on `uvc open` is invoked.

`uvc state` *`devid`*

Returns the image capture state of the device identified by *`devid`*. The result is the string `capture` if the device is started, `stopped` if the device is stopped, or `error` if an error has been detected while image capture was active.

`uvc stop` *`devid`*

Stop capturing images of the device identified by *`devid`*.

`uvc tophoto` *`width height bpp bytearray`* `?`*`rot mirrorx mirrory`*`?`

Makes the RGB (*`bpp`* is 3) or gray (*`bpp`* is 1) byte array *`bytearray`* of *`width`* times *`height`* pixels into a Tk photo image. Optionally, the data is rotated by *`rot`* degrees (possible values 0, 90, 180, 270) and/or mirrored along the X and/or Y axis as specified by the boolean values *`mirrorx`* and *`mirrory`*.

The `uvc` command tries to lazy load Tk, thus allowing to use it from a normal `tclsh`. Only when a photo image is required by a subcommand, Tk must be available and an attempt to load it is made.

v4l2 command

v4l2 command

Name

v4l2 - Video For Linux Two interface

Synopsis

```
package require v4l2
v4l2 option ?arg ...?
```

Description

This command provides several operations to interface Video For Linux Two in order to operate camera devices. *option* indicates what to carry out on the Video For Linux Two subsystem. Any unique abbreviation for *option* is acceptable. The valid options are:

`v4l2 close` *devid*

Closes the device identified by *devid* which has been opened before using `v4l2 open`.

`v4l2 counters` *devid*

Reports a two element list of statistic counters on the device identified by *devid*. The first element is the number of video frames received, the second the number of video frames processed with `v4l2 greyimage` and `v4l2 image`. This information can be used to detect dropped frames.

`v4l2 devices`

Returns a list of device names which can be used for `v4l2 open`. If `udev` support is available, this list is refreshed on plug and unplug of devices. Otherwise it is made up of a snapshot of suitable file names in the `/dev` directory.

`v4l2 greyimage` *devid mask* ?*photoImage*?

Copies the most recent captured image of the device *devid* into the photo image identified by *photoImage* and returns non-zero on success or zero if no data transfer has taken place. The image is converted to greyscale if the capture format delivers color images, where *mask*controls the conversion. If it is empty or **RGB**, all color components are used for conversion, otherwise for each letter **R**, **G**, and **B** the respective color component is used. If more than one color component is used in the conversion, the weights are 0.299 for red, 0.587 for green, and 0.114 for blue. If *photoImage* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel (one or two), and the last the image's pixel values with one or two bytes per grey pixel as a byte array. In this case an error is indicated by throwing an exception.

`v4l2 greyshift` *devid* ?*shift*?

Returns or sets the bit shift to be applied on grey images with a bit depth higher than 8 which are captured from device *devid*. The default value is 4, which is suitable for greyscale cameras with 12 bit resolution. The shift is not applied when the *image* subcommand retrieves raw byte array data.

`v4l2 image` *devid* ?*photoImage*?

Copies the most recent captured image of the device *devid* into the photo image identified by *photoImage* and returns non-zero on success or zero if no data transfer has taken place. If *photoImage* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel, and the last the image's RGB values with 3 bytes per pixel in red, green, blue order as a byte array. In this case an error is indicated by throwing an exception.

`v4l2 info` *devid*

Returns information on open devices. If *devid* is specified, a list of two elements is returned, the first being the device name and the second the image callback command for that device, i.e. the same arguments which were used on `v4l2 open`. If *devid* is omitted, a list of *devid*s, i.e. all currently opened devices is returned.

`v4l2 isloopback` *devname*

Tests if *devname* is a loopback video device and returns true or false.

`v4l2 listen` ?*callback*?

Retrieves or sets the *callback* command called on plug and unplug of devices. When a device is plugged or unplugged that callback is invoked with two additional arguments: the type of event (`add` or `remove`) and

the device name which was added or removed. Only useable if `udev` support is available.

`v4l2 loopback` *devname ?fourcc width height fps?*

Retrieves or sets frame format and rate of the loopback video device *devname*. The parameter *fourcc* specifies the format code, the image size is given as *width* times *height* pixels, and the frame rate *fps* as fraction, i.e. 1/30, or as an integral number, both expressing frames per second. When no parameters are specified, the current settings are returned as a four element list of *fourcc*, *width*, *height*, and *fps*. The supported *fourcc*s are **BGR3**, **BGR4**, **RGB3**, **RGB4**, **GREY**, **YUYV**, and **YVYU**.

`v4l2 mbcopy` *bytearray1 bytearray2 mask*

Copies the content of RGB byte array *bytearray2* into the byte array *bytearray1* using an RGB *mask*. Both byte arrays must have identical length which must be a multiple of 3 (for RGB). The main purpose of this command is to combine images from two cameras into an anaglyph 3D, where (for a red-cyan anaglyph) the left camera image uses mask 0x00FF0000 (red component) and the right camera image uses mask 0x0000FFFF (green and blue components).

`v4l2 mcopy` *photo1 photo2 mask*

Copies the content of the photo image *photo2* into the photo image *photo1* using an ARGB *mask*. Both photo images must have identical width, height, and depth. The main purpose of this command is to combine images from two cameras into an anaglyph 3D, where (for a red-cyan anaglyph) the left camera image uses mask 0x00FF0000 (red component) and the right camera image uses mask 0x0000FFFF (green and blue components).

`v4l2 mirror` *devid ?x y?*

Retrieves or sets flags to mirror captured images along the X or Y axis. Parameters *x* and *y* if specified must be boolean values.

`v4l2 open` *devname callback*

Opens the device with device name (UN*X pathname) *devname* and establishes *callback* as command to be invoked on captured images and returns a *devid*, i.e. a handle to further deal with the device. Two additional parameters are appended when *callback* is invoked: the first is the *devid* of the device, the second a frame counter with initial value of zero based on the last start of image capture. If an error is detected during image capture, the word **error** is used instead of the frame counter.

`v4l2 orientation` *devid ?degrees?*

Retrieves or sets the orientation of captured images regarding image rotation. *degrees* if specified must be an integer number.

`v4l2 parameters` *devid ?key value ...?*

Returns or changes device parameters for the device identified by *devid* given as key-value pairs, e.g. `frame-size 320x240` will change the size of captured images to width 320 and height 240. The command returns the current device parameters (after the potential change, when keys and values where given) as a key-value list which can be processed with `array set` or `dict get`.

`v4l2 start` *devid*

Starts capturing images of the device identified by *devid*. When an image is ready, the callback command set on `v4l2 open` is invoked.

`v4l2 state` *devid*

Returns the image capture state of the device identified by *devid*. The result is the string **capture** if the device is started, **stopped** if the device is stopped, or **error** if an error has been detected while image capture was active.

`v4l2 stop` *devid*

Stop capturing images of the device identified by *devid*.

`v4l2 tophoto` *width height bpp bytearray ?rot mirrorx mirrory?*

Makes the RGB (bpp is 3) or grey (bpp is 1) byte array *bytearray* of *width* times <*height* pixels into a Tk photo image. Optionally, the data is rotated by *rot* degrees (possible values 0, 90, 180, 270) and/or mirrored along the X and/or Y axis as specified by the boolean values *mirrorx* and *mirrory*.

`v4l2 write` *devid bytearray*

Writes the RGB or grey bytes in *bytearray* to the device identified by *devid* which must be an open loopback video device. The size of *bytearray* must match the video width/height of the loopback device. Data is converted to **YUYV** if this is detected for the output path of the loopback device.

`v4l2 writephoto` *devid photo*

Writes the content of the photo image *photo* to the device identified by *devid* which must be an open

loopback video device. The format written is either **RGB4**, **YUYV**, or **GREY** with the photo's dimensions depending on which format is detected for the output path of the loopback device.

The `v4l2` command tries to lazy load Tk, thus allowing to use it from a normal **tclsh**. Only when a photo image is required by a subcommand, Tk must be available and an attempt to load it is made.

For the *fourcc* format codes in `v4l2 loopback`, consult the Linux header file `/usr/include/linux/videodev2.h`. The most useful formats are **RGB4** (8 bits per color in a 32 bit value per pixel), **RGB3** (8 bits per color packed into 24 bits), and **GREY** (8 bits greyscale). Limited support for **YUYV** and **YVYU** (YUV 4:2:2 interleaved) exists for the **v4l2 writephoto** subcommand, too.

wmf command

wmf command

Name

wmf - Tcl interface to cameras using Windows Media Foundation

Synopsis

```
package require tclwmf
wmf option ?arg ...?
```

Description

This command provides several operations to interface cameras using the infrastructure provided by Windows Media Foundation. *option* indicates what to carry out on the Windows Media Foundation subsystem. Any unique abbreviation for *option* is acceptable. The valid options are:

`wmf close` *devid*

> Closes the device identified by *devid* which has been opened before using `wmf open`.

`wmf devices`

> Returns a list of device names which can be used for `wmf open`. Each device adds two elements to the list: its symbolic link to be used in `wmf open` and its friendly name for presentation.

`wmf format` *devid* ?*index*?

> Returns or changes the media format of the device identified by *devid*. The optional parameter *index* is an integer number giving the index of the media format to be used as returned in `wmf listformats`. If omitted, the currently active index is returned. Changing the media format is only possible if the device is not capturing images.

`wmf greyimage` *devid* ?*photoImage*?

> Copies the most recent captured image of the device *devid* into the photo image identified by *photoImage* and returns non-zero on success or zero if no data transfer has taken place. The photo image is filled with grey values. If *photoImage* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel, and the last the image's grey values as a byte array. In this case an error is indicated by throwing an exception.

`wmf image` *devid* ?*photoImage*?

> Copies the most recent captured image of the device *devid* into the photo image identified by *photoImage* and returns non-zero on success or zero if no data transfer has taken place. If *photoImage* is omitted, a four element list is returned with the first element being the image width, the second the image height, the third the number of bytes per pixel, and the last the image's RGB values with 3 bytes per pixel in red, green, blue order as a byte array. In this case an error is indicated by throwing an exception.

`wmf info` *devid*

> Returns information on open devices. If *devid* is specified, a list of two elements is returned, the first being the device symbolic link and the second the image callback command for that device, i.e. the same arguments which were used on `wmf open`. If *devid* is omitted, a list of *devid*s, i.e. all currently opened devices is returned.

`wmf listformats` *devid*

> Returns a dictionary keyed by a media format index as integer with the values being another dictionary with information about the frame size and rate of that media format. The respective index can be used in `wmf format`.

`wmf mbcopy` *bytearray1 bytearray2 mask*

> Copies the content of RGB byte array *bytearray2* into the byte array *bytearray1* using an RGB *mask*. Both byte arrays must have identical length which must be a multiple of 3 (for RGB). The main purpose of this command is to combine images from two cameras into an anaglyph 3D, where (for a red-cyan anaglyph) the left camera image uses mask 0xFF0000 (red component) and the right camera image uses mask 0x00FFFF (green and blue components).

`wmf mcopy` *photo1 photo2 mask*

> Copies the content of the photo image *photo2* into the photo image *photo1* using an ARGB *mask*. Both photo images must have identical width, height, and depth. The main purpose of this command is to

combine images from two cameras into an anaglyph 3D, where (for a red-cyan anaglyph) the left camera image uses mask 0x00FF0000 (red component) and the right camera image uses mask 0x0000FFFF (green and blue components).

`wmf mirror` *devid* ?*x y*?

Retrieves or sets flags to mirror captured images along the X or Y axis. Parameters *x* and *y* if specified must be boolean values.

`wmf open` *devname callback*

Opens the device with device symbolic link *devname* and establishes *callback* as command to be invoked on captured images and finally returns a *devid*, i.e. a handle to further deal with the device. Two additional parameters are appended when *callback* is invoked: the *devid* of the device and the current capture state as in `wmf state`. For the format of *devname* see the description of `wmf devices`.

`wmf orientation` *devid* ?*degrees*?

Retrieves or sets the orientation of captured images regarding image rotation. *degrees* if specified must be an integer number.

`wmf parameters` *devid* ?*key value ...*?

Returns or changes device parameters for the device identified by *devid* given as key-value pairs, e.g. `brightness 100` will change the brightness setting of captured images to the device dependent value 100. The command returns the current device parameters (after the potential change, when keys and values were given) as a key-value list which can be processed with `array set` or `dict get`.

`wmf record` *devid* `frame` *width height bpp bytearray*

Transcodes the frame described by *width*, *height*, *bpp*, and *bytearray* to JPEG and writes the result to the recording file or stream. The recording must have been started with the `-user` option. The bits per pixel parameter *bpp* must be 3. An integer number is returned as result: 1 indicates successful write, 0 no write due to frame rate constraints, and -1 an error during the write.

`wmf record` *devid* `pause`

Pauses recording to a file or stream.

`wmf record` *devid* `resume`

Continues recording to a file or stream.

`wmf record` *devid* `start` *options ...*

Starts recording to a file or stream. *options* control the data format, frames per second, and output channel. The option `-fps` specifies the approximate rate in frames per second as a floating point number. The option `-chan` specifies the channel to which the frames are written. This channel is detached from the Tcl interpreter and controlled solely by the `wmf record` command. The `-boundary` option specifies a MIME multipart boundary string and selects the MIME type **multipart/x-mixed-replace** suitable for streaming to a web browser. The content type delivered to the browser is **image/jpeg**. If the `-boundary` option is omitted, the output format is raw AVI and requires the channel to be seekable. The option `-mjpeg` forces the recorded data to JPEG format, i.e. a transcoding to JPEG will be performed in software, if the device doesn't already deliver a JPEG stream. The option `-user` turns off automatic frame write operations to the recording file or stream when a frame is delivered from the device. Instead, `wmf record` *devid* `frame` must be invoked in the callback function. The `-user` option implies `-mjpeg`.

`wmf record` *devid* `state`

Returns the current recording state as **stop**, **recording**, **pause**, or **error**. The state **error** indicates a write error on the file or stream. In this case no further frames will be written.

`wmf record` *devid* `stop`

Finishes recording to a file or stream and closes the underlying channel.

`wmf start` *devid*

Starts capturing images of the device identified by *devid*. When an image is ready, the callback command set on `wmf open` is invoked.

`wmf state` *devid*

Returns the image capture state of the device identified by *devid*. The result is the string **capture** if the device is started, **stopped** if the device is stopped.

`wmf stop` *devid*

Stop capturing images of the device identified by *devid*.

`wmf tophoto` *width height bpp bytearray* ?*rot mirrorx mirrory*?

Makes the RGB (*bpp* is 3) or grey (*bpp* is 1) byte array *bytearray* of *width* times *height* pixels into a Tk photo image. Optionally, the data is rotated by *rot* degrees (possible values 0, 90, 180, 270) and/or mirrored along the X and/or Y axis as specified by the boolean values *mirrorx* and *mirrory*.

zbar command

Name

zbar::* - interface to the ZBar barcode scanner library.

Synopsis

```
package require zbar
zbar::decode ?options?
zbar::async_decode ?options?
zbar::symbol_types
```

Description

These commands are used to scan barcodes off pixel image data.

`zbar::decode` *photoEtc* ?*syms*?

> Scans the photo image *photoEtc* for barcode information. Alternatively, *photoEtc* can be a four element list describing a greyscale or RGB image as a byte array with 8 bits per color component. The elements must be width, height, number of color components and byte array of the image's pixels in this order. The optional parameter *syms* must be a list of barcode symbologies to be scanned for. If omitted, all known symbologies are tried. The command returns a three element list with the first element being the number of milliseconds spent on decoding. The second element is the decoded symbology on success or an empty string on failure, and the last element is the scan result as a byte array.

`zbar::async_decode` *photoEtc callback* ?*syms*?

> Similar to `zbar::decode` but the decoder is run as a background thread and the result is presented to a *callback* procedure. It requires the Tcl core being built with thread support, and a running event loop since the callback is invoked as an event or do-when-idle handler. Three additional arguments are passed to *callback*: the number of milliseconds for decoding, the decoded symbology on success or an empty string on failure, and the scan result as a byte array. The optional parameter *syms* has the same meaning as in the `zbar::decode` command. Caution: only a single thread instance is supported per Tcl interpreter, i.e. another asynchronous decode process can only be started when a previous decode process has finished.

`zbar::async_decode status`

> Returns the current state of the asynchronous decode thread as a string: *stopped* when no asynchronous decode thread has been started, *running* when a asynchronous decode is in progress, and *ready* when the next asynchronous decode can be started.

`zbar::async_decode stop`

> Stops the background thread for asynchronous decoding if it has been implicitely started by a prior `zbar::async_decode`. This can be useful to conserve memory resources.

`zbar::symbol_types`

> Returns a list of supported symbologies of the scanner, currently `EAN8`, `UPCE`, `ISBN10`, `UPCA`, `EAN13`, `ISBN13`, `DATABAR`, `DATABAR_EXP`, `I25`, `CODABAR`, `CODE39`, `QRCODE`, `CODE93`, and `CODE128`.

ZIP virtual file system

ZIPFS

AndroWish comes with a special ZIP virtual file system which uses mmap(2) to read-only map a ZIP file (in this case AndroWish's APK, i.e. its own installation package) into the process address space to speed up startup time and subsequent read accesses. While this file system was designed primarily for AndroWish it can be used on other platforms, too. Namely, undroidwish uses it on Windows and Linux to mount an archive of Tcl and native extensions which is appended to the executable portion of its binary. It is implemented in the files `zipfs.c` and `zipfs.h` in AndroWish's `.../jni/tcl/generic` folder and enabled in the Tcl core by the presence of the C preprocessor macro `ZIPFS_IN_TCL`.

Low-level C interface

`Tclzipfs_Init(Tcl_Interp *interp)`

> Performs one-time initialization of the file system and registers it process wide. Additionally, a package named *zipfs* is provided and supplemental Tcl commands are created in the given interpreter.

`Tclzipfs_Mount(Tcl_Interp *`*`interp`*`, const char *`*`zipname`*`, const char *`*`mntpt`*`, const char *`*`passwd`*`)`

> Mounts the ZIP archive file *`zipname`* on the mount point *`mntpt`* using the optional ZIP password *`passwd`*. Errors during that process are reported in the interpreter *`interp`*. If *`zipname`* is a NULL pointer, information on all currently mounted ZIP file systems is written into *`interp`*'s result as a sequence of mount points and ZIP file names.

`Tclzipfs_MountBuffer(Tcl_Interp *`*`interp`*`, const char *`*`mntpt`*`, unsigned char *`*`data`*`, int `*`length`*`, int `*`copy`*`)`

> Mounts the ZIP archive contained in the memory buffer described by *`data`* and *`length`* on the mount point *`mntpt`*. Depending on *`copy`* a private copy of this memory buffer is made and used for the mount operation. Errors during that process are reported in the interpreter *`interp`*. If the mount operation succeeds, a string of the form "`memory_<size>_<id>`" is left in *`interp`*'s result identifying the archive from the memory buffer. This information is useful as `zipname` parameter in a later unmount operation. If *`mntpt`* is a NULL pointer, information on all currently mounted ZIP file systems is written into *`interp`*'s result as a sequence of mount points and ZIP file names.

`Tclzipfs_Unmount(Tcl_Interp *`*`interp`*`, const char *`*`zipname`*`)`

> Undoes the effect of `Tclzipfs_Mount()`, i.e. unmounts the mounted ZIP archive file *`zipname`*. Errors are reported in the interpreter *`interp`*.

Tcl commands

The `zipfs` package provides Tcl with the ability to mount the contents of a ZIP file as a virtual file system.

`zipfs::exists `*`filename`*

> Return 1 if the given *`filename`* exists in the mounted zipfs and 0 if it does not.

`zipfs::find `*`dir`*

> Recursively lists files including and below the directory *`dir`*. The result list consists of relative path names starting from the given directory. This command is also used by the `zipfs::mkzip` and `zipfs::mkimg` commands.

`zipfs::info `*`file`*

> Return information about the given *`file`* in the mounted zipfs. The information consists of (1) the name of the ZIP archive file that contains the file, (2) the size of the file after decompression, (3) the compressed size of the file, and (4) the offset of the compressed data in the ZIP archive file.
>
> Note: querying the mount point gives the start of ZIP data offset in (4), which can be used to truncate the ZIP info off an executable.
>
> Note: the file of a mounted ZIP archive appears as directory but can be opened and read like a regular file if the mount process detected a non archive area in front of the ZIP archive, e.g. when the ZIP archive was appended to an executable file. In this case that area can be read using the Tcl `open` and `read` commands but `file copy` treats the mounted archive as a directory.

`zipfs::list ?-`*`glob`*`|-`*`regexp`*`? ?`*`pattern`*`?`

> Lists files of any or all of the mounted ZIP archives. If *`pattern`* is omitted all files are listed. Otherwise *`pattern`* is interpreted as a glob or regexp pattern and used to list only files matching this pattern.

`zipfs::lmkimg `*`outfile inlist ?password? ?infile?`*

Like `zipfs::mkimg` but instead of an input directory *`inlist`* must be a list where the odd elements are the original input file names as copied into the archive and the even elements their respective names within the archive.

`zipfs::lmkzip` *`outfile inlist`* ?*`password`*?

Like `zipfs::mkzip` but instead of an input directory *`inlist`* must be a list where the odd elements are the original input file names as copied into the archive and the even elements their respective names within the archive.

`zipfs::mkimg` *`outfile indir`* ?*`strip`*? ?*`password`*? ?*`infile`*?

Create an image (potentially a new executable file) similar to `zipfs::mkzip`. If the *`infile`* parameter is specified, this file is prepended in front of the ZIP archive, otherwise the file returned by `Tcl_NameOfExecutable(3)` (i.e. the executable file of the running process) is used. If the *`password`* parameter is not empty, an obfuscated version of that password is placed between the image and ZIP chunks of the output file and the contents of the ZIP chunk are protected with that password.

Caution: highly experimental, not usable on Android, only partially tested on Linux and Windows.

`zipfs::mkkey` *`password`*

For the clear text *`password`* argument an obfuscated string version is returned with the same format used in the `zipfs::mkimg` command.

`zipfs::mkzip` *`outfile indir`* ?*`strip`*? ?*`password`*?

Creates a ZIP archive file named *`outfile`* from the contents of the input directory *`indir`* (contained regular files only) with optional ZIP password *`password`*. While processing the files below *`indir`* the optional prefix given in *`strip`* is stripped off the beginning of the respective file name.

Caution: the choice of the *`indir`* parameter (less the optional *`strip`* prefix) determines the later root name of the archive's content.

`zipfs::mount` ?*`zipfile`* ?*`mountpoint`*? ?*`password`*?

`zipfs::mount -file` *`zipfile mountpoint`* ?*`password`*?

`zipfs::mount --` *`zipfile mountpoint`* ?*`password`*?

This command mounts a ZIP archive file as a VFS. After this command executes, files contained in *`zipfile`* will appear to Tcl to be regular files at the mount point.

In the first command form, with no *`mountpoint`*, returns the mount point for *`zipfile`*. With no *`zipfile`*, return all zipfile/mount point pairs. If *`mountpoint`* is specified as an empty string, the mount point will be the current directory. If *`password`* is specified, files from *`zipfile`* are decrypted using this password when read.

`zipfs::mount -data` *`bytearray mountpoint`*

The data in *`bytearray`* must represent a ZIP archive which gets mounted on *`mountpoint`*. If the mount operation succeeds, the result is a string of the form "`memory_<size>_<id>`" which can later be used as *`zipfile`* parameter in an unmount operation.

`zipfs::mount -chan` *`channelId mountpoint`*

A ZIP archive is read from channel *`channelId`* and mounted on *`mountpoint`*. If the mount operation succeeds, the result is a string of the form "`memory_<size>_<id>`" which can later be used as *`zipfile`* parameter in an unmount operation.

`zipfs::unmount` *`zipfile`*

Unmounts the mounted ZIP archive file *`zipfile`*.

`zipfs::unwrap` *`?filename?`*

If *`filename`* is the root of a mounted ZIP archive its content is unpacked to a local directory named `filename.vfs`. This directory must not exists prior to the call. Otherwise, *`filename`* is temporarily mounted before the unpack operation takes place and unmounted afterwards. If *`filename`* is omitted the result of `info nameofexecutable` is used instead, i.e. the main ZIP archive of the running process is unpacked.

The commands described above are available as subcommands in the `zipfs` ensemble, i.e. `zipfs list` is equivalent to `zipfs::list`.

zipfs as Tcl (and Tk) bootstrap file system

On the Android platform zipfs is used to boot Tcl/Tk from the APK by early mounting the APK file on the file system root as seen by Tcl. Since nearly all relevant files within the APK are below the `assets` folder, this lets Tcl see the directory `/assets` with its library directories, e.g. the `/assets/tcl8.6` directory with Tcl's library modules, encoding tables etc. That relationship to `/assets/tcl8.6` is hard coded into the Tcl shared library and based on it all other packaged library directories can be found during Tcl initialization.

For standalone apps a similar approach is chosen by hard coding the file `/assets/app/main.tcl` as the file to be sourced (if present) right after Tcl's initialization. This allows for packaging Tcl based apps as an APK, see the description in AndroWish SDK for instructions.

On other platforms (currently tested Linux and Windows) the initial mount of an embedded ZIP file system is done on the executable itself, e.g. if `/home/john/awish` is the Tcl/Tk binary with an included ZIP file system, the Tcl library directory of the file system when mounted becomes `/home/john/awish/tcl8.6`. Similarly, built in application code will be started from the file `/home/john/awish/app/main.tcl` if present. Additionally, the contents of the optional file `/home/john/awish/app/cmdline` are appended to the command line before Tk is initialized and control is transferred to the `main.tcl` script. This is useful to setup certain aspects of SDL, e.g. to start in full screen mode with or without changed display resolution (see description of SDL startup options in Beyond AndroWish). Another hook is `/home/john/awish/app/icon.bmp` which (if present) should be a Windows BMP 24 bit RGB bitmap file used as the icon for the SDL root window.

On Windows platforms the drive letter of the base executable is prepended to the respective path names. For the example above this means: `C:\home\john\awish.exe` is the binary, `C:/home/john/awish.exe/tcl8.6` becomes the Tcl library directory, `C:/home/john/awish.exe/app/main.tcl` is the optional application script, and so on.

For a small sample script refer to Make minimal vanillawish binary.

Some delicate implementation details

For loading binary Tcl extensions (shared libraries) on certain platforms (Linux and FreeBSD) special handling is tried to be carried out:

- on Linux, the **memfd_create** system call is used (if available) to make a memory backed file with the payload in the **/dev/shm** namespace which finally is **dlopen**'ed to provide the shared library.
- on FreeBSD, there's **fdlopen** which allows a file descriptor to be treated as a shared library. Very similar to the Linux approach, the file descriptor refers to a **/dev/shm** memory backed file which is primed with the contents for the shared library from the ZIP archive.

For improving **glob** operations the ZIP virtual file system uses two hash based data structures: **ZipEntry** for regular files and **ZipDirEntry** for directories which additionally contains a hash table to accelerate lookups in this directory. For typical searches, this usually outperforms the native OS functions.