TCL - TK Form Generator

Disclaimer

This SOFTWARE PRODUCT is provided by El Condor "as is" and "with all faults." El Condor makes no representations or warranties of any kind concerning the safety, suitability, lack of viruses, inaccuracies, typographical errors, or other harmful components of this SOFTWARE PRODUCT. There are inherent dangers in the use of any software, and you are solely responsible for determining whether this SOFTWARE PRODUCT is compatible with your equipment and other software installed on your equipment. You are also solely responsible for the protection of your equipment and backup of your data, and El Condor will not be liable for any damages you may suffer in connection with using, modifying, or distributing this SOFTWARE PRODUCT.

You can use this **SOFTWARE PRODUCT freely, if you would you can credit me in program comment:**

El Condor – CONDOR INFORMATIQUE – Turin

Comments, suggestions and criticisms are welcomed: mail to rossati@libero.it

Conventions

Commands syntax, instructions in programming language and examples are with font `COURIER NEW`. The optional parties of syntactic explanation are contained between `[square parentheses]`, alternatives are separated by `|` and the variable parties are in *`italics`*.

Contents table

1 Form generator

Form generator, briefly *TK FormGen*, is a set of Tcl scripts which allows to build and handle forms data; it is sufficiently generalized for a wide use.

The form generator (*TK FormGen*) is composed by two scripts:

- `fGen.tcl` that contains the `proc`edure for generate a form,
- ~~`fGenProc.tcl` that contains some procedures and is automatically included by `fGen.tcl`;~~

1.1 Using the form generator

Requires Tcl 8.5 or later.

1.1.1 Call syntax

`TK FormGen` can be invoked by a Tcl script or directly executing `fGen.tcl` with arguments:

- `formGen [list` *`parms|fileParms`* `[`*`outputFile|callBackFunction`*`] [`*`handleEvents`*`]]`
- `tclsh fGen.tcl` *`parms|fileParms`* `[`*`outputFile|callBackFunction`*`] [`*`handleEvents`*`]`

The first syntax is for invoke *`TK FormGen`* from a Tcl script, the second when it is invoked from command line or by another language.

`parms|fileParms` this parameter is a string with the description of widgets or a file that contains the description of widgets.

`outputFile|callBackFunction` is the eventual parameter which deals on the close of form; it can be a procedure which receive the data or a name of an output file where the data is written; see the paragraph 1.2.8 Returned Values for how data are obtained. and paragraph 1.3.2 Handle CallBack.

`handleEvents` is the eventual `proc`edure name for manage widgets events (see 1.3.1 Handle Events).

1.1.2 How *TK FormGen* can be called

The form generation can be called:

- by a tcl script,
- directly from command line,
- by another language, see chapter *3 Using with programming languages*.

In the first alternative a Tcl script must include `fGen.tcl` (by command `source`) and call the generator:

```
proc cBack {data} {
       foreach id [dict keys $data] {puts "$id\t : [dict get $data $id]"}
       exit
}
append src [file dirname [file normalize [info script ]]] "/fGen.tcl"
source $src
set ask "Title,Form generalities;Ground,silver;"
append ask "CMB,Theme,,,|alt|clam|classic|default|vista|winnative|xpnative;"
append ask "CMB,Ground,,,|cyan|gray|magenta|olive|silver|teal;"
append ask "CMB,Form,Form type,,|T:Texts|B:Buttons|C:Combo boxes|F:Files and folders|
S:Scales and combos;"
append ask "Default,Theme:clam,Ground:teal;"
formGen [list $ask cBack]
```

```
C:\ActiveTcl\source>tclsh fgen.tcl "T,name,User name;P,psw,Enter password" data.json

C:\ActiveTcl\source>type data.json
{"name": "La Souris",
"psw": "Lafayette3.14",
"fg_Button": "Ok"}
```

```
C:\ActiveTcl\source>
```

☞ If the *outputFile* suffix is `.json` the data are built in JSON format.

~~If *CallBackFunction* and *handleEvents* are in another script they must be invoked with the syntax: *script*.tcl::*procName*:~~

~~`tclsh fGen.tcl params.txt handler.tcl::tryForm handler.tcl::handle`~~

1.2 Data description

Every widget is characterized by a list of attributes, comma separated, in this order: widget Type, Field Name, Field Label, Length and Extra(s). Widgets are separated by semicolon.

In addition to the Widgets there can be some others information (*Pseudo types*) with different semantics that will be detailed in the paragraphs dedicated to them.

If the widget list starts with `#` it is a comment which ☞ must also be terminated by semicolon.

1.2.1 Widget type

☞ The Types are indifferent to the case.

- Buttons:
  - **B** button;
  - **R** radio button, a set of radio buttons;
- **CHECK**, **CKB** check box;
- Combo boxes:
  - **CMB** combo box;
  - **CMT** is a combo box with Text associated for insert values not in combo;
- Text fields:
  - **A** multi line text,
  - **N** numeric field;
  - **DN** decimal numeric field;
  - **F**, **FW** e **D** file and directory;
  - **IN** absolute integer;
  - **H** or **HIDDEN** hidden field;
  - **SCALE, S** scale (or slider);
  - **QS** qualitative scale;
  - **Password**, **P** password field, the data entered are masked;
  - **Text**, **T** text field;
  - **Time**;
  - **U**, **UN** not modifiable fields, **UN** is a not modifiable numeric field.

1.2.2 Widget Name

Is the name of the field that, when the form is closed, is the key of the `dict`ionary returned with the values.

☞ The name is case-sensitive.

1.2.3 Widget Label

If omitted the widget *label* or the button caption it is used the Field *Name*.

1.2.4 Widget Length

The *length* of the Widget; it can be, depending on the type of widget, in pixel characters or lines. If it is omitted it is assumed depending on the *type* of Widget:

	Widget type	**Value** (characters if unspecified)
A	Area text	5 lines
D	Folder	30
DN	Decimal numeric text	10
F WF	File write File	40
IN	Numeric unsigned text	7
N	Numeric text	8
QS	Qualitative scale	150 pixel
S	Scale	180 pixel
P	Password	16
R	Radio button	12
T	Text	20

☞ if the length of Text is greater of 50, it is treated like an Area type.

1.2.5 Extra(s)

Extra Field is used for add information to the widget.

Widget	**Type**	**Extra field(s)**
Button	**B**	Possibly name of CallBack function
Check box	**CKB**	Possible Description at right of check box
Combo box	**CMB**	An item list separated by | whit form [*key*:]*value*
File	**F, FW**	File type(s)
Radio button	**R, RDB**	an item list separated by | whit form [*key*:]*value*
Scale	**S,SCALE**	Start, end and eventual precision value, default is 0 100 0
Qualitative scale	**QS**	an item list separated by |
Texts	**T,IN,DN ,N,P**	Tooltip (*)

(*) if package `tooltip` is not accessible the tool-tips are ignored.

1.2.6 Summary by type

1.2.6.1 Buttons

The package adds the standard buttons `Ok`, `Cancel` and `Reset`; they have the names respectively `fg_Ok`, `fg_Cancel` and `fg_Reset`.

The *extra* field can contains a name of the function called when the Button is pushed,.

The Ok button is replaced if there is almost one type **B** widget in the list not associated by `AFTER` pseudo type to some control.

The caption of standard buttons can be modified inserting a **B** type widget with this syntax:

```
B,[fg_Ok|fg_Cancel|fg_Reset],caption,[...
```

The button caption can be a name of an image or a Unicode character which is a simple and efficient way to create buttons with pictures: the Unicode characters is accepted in the form:
`\ueeee` or `\xee` where `ee`(`ee`) is a hexadecimal value of the Unicode character.

```
B,fg_Cancel,\u2718;
B,fg_Reset,\u21B6;
B,Start,\u270E;
```

Table 1: Some UNICODE characters

Name	Decimal value	Symbol	Hexadecimal value
edit	9998	✎	\u270E
delete	10008	✘	\u2718
check	10003	✓	\u2713
check bold	10004	✔	\u2714
Left circular arrow	8630	↶	\u21B6
email	9993	✉	\u2709
cross	10006	✖	\u2716
dollar		$	\x24
euro		€	\u20AC
pound		£	\xA3
white square		▢	\u25a2
triangle down	9660	▼	\u25bc

1.2.6.2 Check box

The Check box value is 1 for checked and 0 for unchecked.
The *extra* field can contains a possible description after the check box.
The value returned of check box is 0 or 1 or empty if it has not checked nor set by *default* command.

1.2.6.3 Date

The `Date` widget permits to insert a date; the *extra* field can contain a tips and the *extra2* field can contain the format (the default is `yyyy/mm/dd`).

1.2.6.4 File, save file and folder

The initial Folder (and possibly file name) can be set by *default* pseudo type. The *extra*(s) field of type **F**ile and **W**rite **F**ile can be a file type in the form:

description: *extension*[*extension* ...]

example: `File,File,,40,Image Files:*.gif *.jpg,Text Script:*.tcl,All: *;`

1.2.6.5 Radio buttons

The *extra* field contains the item list separated by `|`. For get a key instead the description, the item must have the form: *key*:*value*.
The *default* value must be the *key* or *value* if the *key* not exists.

```
Rdb,Status,,,M:Married|S:Single|W:Widow;
Rdb,AgeType,Age type,,M:Months|Y:Years;
Default,Status:S,AgeType:M;
```

It is possible to have more than one set of radio buttons.

1.2.6.6 Scale

The *length* is the length in pixels.
The *extra* field of the type **S** can contains the `start`, `end` and

abs(start - end)	n. decimals

possibly `precision` values in the form `start end precision`, e.g. `-5 5 1`; if omitted the range is `0 100` and precision `0`; the result can have decimals depending on the difference from `start` and `end` value, see table at right; `start` can be greater of `end` e.g.:

```
Scale,lg,Loss-Gain,,10 -10
```

If precision is negative the value returned is divided by $10^{-precision}$.

> 99	0
<100 and > 10	1
<10 and > 1	2
<1 and > 0.1	3
...	...

The qualitative scale (type **QS**) returns qualitative values taken from the *extra* field where they have the same syntax of the values of radio buttons:

```
...
QS,Urgency,,100,White|Green|Yellow|Red
...
```

1.2.6.7 Combo box

CMB is a simple combo box, if no member is selected the value returned is an empty string; if the form has only one combo, there aren't buttons and the form is exited when an item is selected.

The *extra* field of combo contains the item list separated by `|` (see description in Radio button).

CMT type is a combo that allows to insert a value not present in the list.

The returned value for **CMB** and **CMT** is a key, if present otherwise is the value selected.

1.2.6.8 Text fields

The possibly *extra* field is the Tool-tip.

☞ The control of numeric type (**N**, **IN**, **DN**) doesn't allows multiple leading zeroes[1].

1.2.6.9 Time

The format accepted is `hh:mm:ss`.

1.2.7 Pseudo types

Pseudo fields are flavors for show form; the syntax is different from the normal widgets.

1.2.7.1 After

The pseudo field `after` is useful for insert a widget at right of another widget:

```
after,WidgetNameA,WidgetNameL
```

where *`WidgetNameA`* is the Widget to be placed at right of the Widget *`WidgetNameL`*

```
T,Mail,E mail,18;
RDB,Gender,,,M:Male|F:Female|U:Unspecified;
B,infoButton,✔;
After,infoButton,Mail;
```

☞ In the list of wiews *`WidgetNameL`* must appears before *`WidgetNameA`*. and both must precede the `After` pseudo type.

1.2.7.2 Comment

[C|Comment],*comment*

The comment occupies the space of label and widget.

1.2.7.3 Ground, Background

```
[ground|background],[backgroundColor|Gray],[theme]
```

[1] For JSON compatibility.

The pseudo field `Ground` or `Background` sets the form background.
Tk recognizes many symbolic color names, see https://www.tcl.tk/man/tcl8.5/TkCmd/colors.htm.
The colors can be in hexadecimal notation: **#**`RGB` or **#**`RRGGBB`.
`theme` is one of themes disposable, default is `winnative`.

☞ `puts [ttk::style theme names]` list the disposable themes: `winnative clam alt default classic vista xpnative`.

1.2.7.4 Validate

This pseudo field **V** or **Validate** is used for some controls on fields:

```
validate,fieldName operator (value|fieldName)[,errorMessage]
```

where operator can be one `[[greater|gt|>]|is]`.

After the operator `is` we can have:

```
required|mail|regularExpression
...
set p "T,email,My EMail,20;"
append p "P,password,type password;"
append p "P,repeatPassword,retype password;"
append p "validate,email is mail,Incorrect mail form;"
append p "validate,password=repeatPassword;"
append p "Validate,psw,is .{6#44},Password too short;"
...
```

☞ if *`value`* contains comma or semicolon they must be masked (see paragraph 1.4.1).

1.2.7.5 Default

The syntax is: `default[s],`*`WidgetName`*`:`*`WidgetValue`*`[,...]`

`defaults` is useful for populate the form.

☞ if *`WidgetValue`* contains comma or semicolon they must be masked.

☞ The default for combo can be both one value exposed or one key.

1.2.7.6 Hidden field

Is the type **`H`** or **`HIDDEN`**; the syntax is: `[H|Hidden],`*`fieldName`*`,`*`value`* es:

```
set p "Title,Example of form;"
...
set systemTime [clock seconds]
append p "H,hField,[clock format $systemTime];"
formGen $p "cBack2"
```

1.2.7.7 Label(s)

For manage the widgets label:

```
label,right|left,afterLabel
```

`afterLabel` are the possibly characters inserted after the label.

ex. `Ground,silver;Title,,Label example;Label,Right, :;`

1.2.7.8 Required

A list of widgets to which a value must be given:

```
req[uired],field1[,field2...]]
```

1.2.7.9 Title

`Title,`*`formTitle`*

Ex. `Title,Send mail parameters;`

The default title is Tcl - Tk Form Generator.

1.2.8 Returned Values

The format of the data when a form is closed depends on the third parameter of the call:

the parameter is absent	data are sent to standard output in Json format
the parameter is a call back function	call back function receive the data in a `dict`ionary
the parameter is a json file	data are written in Json format (*)
the parameter is a file	the data are written one for line with format `name value`

(*) The value of numeric data, i.e. numeric text and seek bar, if not present is set to `0`.
In addition to the data the is present the field `fg_Button` which contains the name of the button pushed (`Ok` or `Cancel` or the name of custom button or the name of the lonely combo).

☞ If `Cancel` button was pressed there is only `fg_Button` field.

1.3 CallBack

TK FormGen works on CallBack not only at the end of form, but also, possibly, for handle events (the fourth parameters of the call), or by a `proc`edure associated to a **B** button (the *extra* field). The CallBack function receive the button name.

1.3.1 Handle Events

This function can be used to personalize the form e.g. modify the state of the Widget, perform controls end so on. The functions receive a list which contains the name of the event and, depending on event, path, and widget name.

Event	Path	Note
`Start`		
`End`	`Button`	
`Reset`		
`lFocus`	`path`	lost focus.
`Scale`	`path`	
`Select`	`path`	Combo boxes

```
proc handle {e} {
  lassign $e event path name
  if {$event == "Start"} {
    fg_setCombo Theme "[join [ttk::style theme names] "|"]"
  }
  if {$name == "Form" && $event == "Select"} {
    if {[valueOf $name] != ""} {setValue try "[dict get $::handler::examplesMap
[valueOf $name]]"}
  }
}
```

Example of sub for handle events

1.3.2 Handle CallBack

The function is called when a type **B** button with call back function (the *extra* field) is clicked; the function receive the name of the button. The values of Widget can be accessed by the function `valueOf WidgetName`.

```
proc infoMail {button} {
      tk_messageBox -message "Password almost 6 characters" -type ok -icon
info
}
```

Sample of Callback function

1.4 Remarks

1.4.1 Masking comma and semicolon

If *label* or *default* or *extra* or condition contains commas or semicolons they must be coded with their hexadecimal value, respectively `#2C` and `#3b`.

1.4.2 Handling Buttons

TK FormGen inserts the `Ok` button, the `Cancel` button and the `Reset` button depending on the Widgets contained in the form:

- only `Cancel` button if the form contains only `Comment` widgets,
- no buttons if there is only one combo box (**CMB** type) and eventual `Comment`(s) widgets,
- the `Reset` button is present if there are data widgets,
- the `Ok` button is not present if there are some others buttons not aftered.

1.4.3 Data presentation

The data are presented in the order they appears in the parameters list, except for the Type **B** buttons that appears together buttons inserted by *TK FormGen*, at the bottom of the form (if is not aftered).

☞ If the length of type **T**ext widget exceed the maximum characters allowed for the line, the widget is set to `Area` widget.

With the pseudo type `after` buttons or check box can be placed at right of another widget.

1.4.4 Work with Widgets

- Change the value: **`setValue`**`(`*`WidgetName`*`,`*`value`*`)`
- Get the Widget value: **`valueOf`**`(`*`WidgetName`*`)`

1.5 Others functions and utilities

1.5.1 Get and set widget value

- `ValueOf` *`name`* when the form is active
- `getDictValue` *`dataMap Field default`* for access values on the possible function called by event on exit button(s); ex. `Set x [getDictValue $data Store ""]`.
- `setValue` *`name value`* `[`*`extra2`*`]`

1.5.2 Mask comma and semicolon

The procedure `mask` *`data`* change comma and semicolon in string *`data`* with their hexadecimal value, respectively `#2C` and `#3b`.

1.5.3 Left

The procedure `left` *`data length`* returns the first *`length`* characters of *`data`*.

1.5.4 Mright

The `proc`edure `mRight` *`data position`* returns the last characters of *`data`* starting from *`position`*.

1.5.5 Reset form

```
fg_Restore
```

1.5.6 Set combo values

The `proc`edure `fg_setCombo` *`name items`* can be used for set or change the combo values.

See example on paragraph 1.3.1 Handle Events.

1.5.7 Signal

`fg_signal` *`message`* `[`*`title`*`]` `[`*`timer`*`]`

where the possible *`timer`* is in milliseconds.

Generate a timed popup ex.

```
set signal "Added $Qty $Name"
if {[db errorcode] != 0} {set signal "Errore: [db errorcode]"}
fg_signal $signal "FaRo" 2000
```

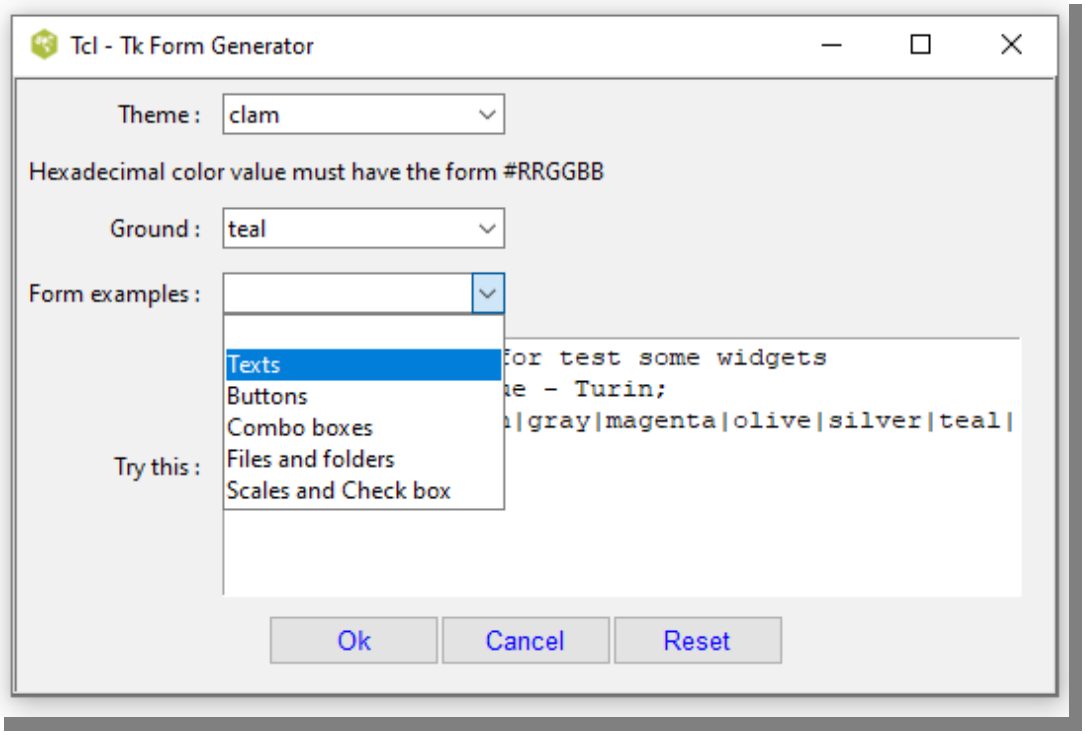
1.6 Customization

frame	`.fg`	`configure -relief [raised|sunken|flat|ridge|solid|groove]`
title	`.fg.title`	

2 Use on TCL applications

3 Using with programming languages

The examples deal on a form (see image) that can be used for test widgets and also for explore different form backgrounds and themes.



When the form is closed the data are sent in Json format to standard output for to be transformed in internal variables of the host language.

Below the list for create the form (file `params.txt`).

```
CMB,Theme;# themes are set at the beginning with the available themes;
Labels,Right, :;
Comment,Hexadecimal color value must have the form #RRGGBB;
CMT,Ground,,,|cyan|gray|magenta|olive|silver|teal|#C0D0E0;
CMB,Form,Form examples,,|Texts|Buttons|Combo boxes|Files and folders|Scales and Check
box;
```

```
T,try,Try this,400;
Default,Theme:clam,Ground:teal;
Validate,Ground is ^#[0-9a-fA-F]{6}$,Not valid color;
Default,try:C#2CThis is a form for test some widgets\nCondor Informatique - Turin#3b
CMB#2CGround#2C#2C#2C|cyan|gray|magenta|olive|silver|teal|#C0D0E0;
```

In all the languages with which *TK FormGen* was tested, the form was generated by executing the TCL interpreter via external command and data was read from the standard output and transformed in internal variables. The command sent is:

```
tclsh fGen.tcl params.txt handler.tcl::tryForm handler.tcl::handle
```

The script `handler.tcl` contains the `proc`edure `tryForm` that receive the data for generate a second form and the `proc`edure `handle` for manage the form events, in particular when an item of the combo box `Form` is selected, it fills the `try` text area with a corresponding form list.

```
namespace eval handler {
      variable examplesMap [dict create
            "Texts" ... \
            "Scales and Check box" ... \
            "Buttons" ... \
            "Combo boxes" ... \]
}
proc handle {e} {
      lassign $e event path name
      if {$event == "Start"} {
            fg_setCombo Theme "[join [ttk::style theme names] "|"]"
      }
      if {$name == "Form" && $event == "Select"} {
            if {[valueOf $name] != ""} {setValue try "[dict get
$::handler::examplesMap [valueOf $name]]"}
      }
}
proc tryForm {data} {
      append prefix "Title,Form example of " [dict get $data "Form"]
";Ground," [dict get $data "Ground"] "," [dict get $data "Theme"]
";Labels,Right, :;"
      set systemTime [clock seconds]
      append prefix "H,hField,[clock format $systemTime];" [dict get $data
"try"]
      formGen [list "$prefix"]
}
```

3.1 C++

Table 2: C++ script

```
// C++ 11
#include <bits/stdc++.h>
using namespace std;
// function to find all the matches
map<string, string> json2Map(string jsonData) {
    map<string, string> dataMap;
    string regExspr("\\\"(\\w+)\\\":\\s*(\\\"([^\"]*)\\\"|(([+-]?\\d*)(\\.\\
d+)?))");
    regex re(regExspr);
```

```
    for (sregex_iterator it = sregex_iterator(jsonData.begin(),
jsonData.end(), re);it != sregex_iterator(); it++) {
            smatch match;
            match = *it;
            int indexValue = 3;
            if (match.str(3) == "") indexValue = 2;
            cout << match.str(1) << "\t" << match.str(indexValue) << endl;
            dataMap[match.str(1)] =  match.str(indexValue);
    }
    return dataMap;
}
int main( int numberArgs, char* argsList[] ) {
    vector<string> params{ "fGen.tcl","T,t1;N,n1", "", "" };
    int commandLength = 40; // roughly default + fixed (i.e. tclsh ...
>j.json)
    for (int i=1; i<numberArgs; i++) {
        params[i-1] = argsList[i];
        commandLength += strlen(params[i-1].c_str());
    }
    char tcl[commandLength];
    sprintf (tcl, "tclsh %s %s %s %s > j.json", params[0].c_str(),
params[1].c_str(),params[2].c_str(),params[3].c_str());
    int status = system(tcl);
    ifstream jsonFile("j.json");
    string jsonData {istreambuf_iterator<char>(jsonFile),
istreambuf_iterator<char>()};
    cout << jsonData << endl;
    map<string, string> dataMap = json2Map(jsonData);
    if (dataMap.count("Integer") > 0) {
        cout << "Sum: " << (atof(dataMap["Integer"].c_str()) +
atof(dataMap["Number"].c_str()) + atof(dataMap["DecNumber"].c_str())) << endl;
    }
      return 0;
}
```

Table 3: Batch command

```
cls
C\tcltk fGen.tcl params.txt handler.tcl::tryForm handler.tcl::handle
```

3.2 NodeJS

Table 4: NodeJS script (`tcltk.js`)

```
const args = process.argv.slice(2)
const {exec} = require('child_process');
var fs = require('fs');
var command = 'tclsh ' + args[0] + ' "' + args[1] + '" ' + args[2] + ' "' +
args[3]
const tcl = exec(command, function (error, stdout, stderr) {
  if (error) {
      console.error(`exec error: ${error}`);
      return;
  }
  console.log('Child Process STDOUT: '+stdout);
});
let jsonData = '';
```

```
tcl.stdout.on('data', (chunk) => {
  jsonData += chunk.toString();
});
tcl.on('exit', () => {
  console.log(jsonData);
  var data = JSON.parse(jsonData)   // the data becomes a JavaScript object
  if ("Integer" in data) {
      console.log("Sum " + (data.Integer + data.Number + data.DecNumber))
      }
});
```

Table 5: Batch command

```
cls
node NodeJS\TclTk.js fGen.tcl params.txt handler.tcl::tryForm handler.tcl::handle
```

3.3 Python

```
#-------------------------------------------------------------------------------
# Name:     tcltk.py
# Purpose:  Use of TK form Generator by Python v3.2.5
# Author:   El Condor
# Created:  10/09/2019
#-------------------------------------------------------------------------------
import os
import subprocess
import json
def main():
    pass
os.chdir("C:\\Sviluppo\\fGenTk\\")
p = subprocess.Popen("tclsh fGen.tcl params.txt handler.tcl::tryForm
handler.tcl::handle", shell=True, stdout=subprocess.PIPE)
stdout, stderr = p.communicate()
data = json.loads(stdout.decode('utf-8'))
print(data)
if 'Integer' in data:
   print ("Sum: {}".format(data['Integer'] + data['Number'] +
data['DecNumber']))
if __name__ == '__main__':
    main()
```

```
*** Python 3.2.5 (default, May 15 2013, 23:06:03) [MSC v.1500 32 bit (Intel)]
on win32. ***
>>>
{'fg_Button': 'Ok', 'textArea': 'Text Area\n', 'Text': 'A text', 'DecNumber':
3.1415, 'Number': -11, 'Integer': 7, 'Mail': 'ross@lib.it', 'psw':
'Lafayette3.14'}
Sum: -0.8584999999999998
>>>
```

3.4 R

For this example needs `install.packages("rjson")`.

```
library("rjson")
setwd("C:\\Sviluppo\\fGenTk")
parameters <-
c("fGen.tcl","params.txt","handler.tcl::tryForm","handler.tcl::handle")
```

```
json <- system2("tclsh",parameters,stdout=TRUE)
tmp <- tempfile()
cat(json,file = tmp)
data <- fromJSON(readLines(tmp))
data$fg_Button
if ("Integer" %in% names(data))
      cat("Sum:",data$Integer + data$Number + data$DecNumber,"\n")
```

```
library("rjson")
setwd("C:\\Sviluppo\\fGenTk")
parameters <- c("fGen.tcl", "\"Title,Is triangle?;DN,a;DN,b;DN,c;\"")
json <- system2("tclsh",parameters,stdout=TRUE)
tmp <- tempfile()
cat(json,file = tmp)
data <- fromJSON(readLines(tmp))
sides <- sort(c(data$a,data$b,data$c))
sides
if (data$fg_Button != "Cancel")
if (sides[1] + sides[2] > sides[3])
      print("Is triangle") else print("Is not triangle")
```

3.5 Ruby

```
require 'json'
require 'FileUtils'
FileUtils.cd('c:/Sviluppo/fGenTk')
output = `tclsh fGen.tcl params.txt handler.tcl::tryForm handler.tcl::handle`
data = JSON.parse(output) if output != ""
puts JSON.pretty_generate(data).gsub(":", " =>")
if data.has_key?("Integer") then
      puts sprintf("Sum: %f",data['Integer'] + data['Number'] +
data['DecNumber'])
end
```

```
>ruby tcltk.rb
{
  "fg_Button" => "Ok",
  "Text" => "text boia faus",
  "psw" => "krinass",
  "Mail" => "ross@lib.it",
  "Number" => -11,
  "Integer" => 17,
  "DecNumber" => 3.1415,
  "textArea" => "puts JSON.pretty_generate(my_hash).gsub(\" =>\", \" =>\")\n"
}
Sum: 9.141500
>Exit code: 0
```

3.6 Rust

For this example needs `cargo install json`.

4 Technical notes

Window name `.fg`

4.1 Frames

`$path``.after_`*`$afterName`* frame for add widgets after widgets.

4.2 Widget names

widgets field name	*.frm.name*	*name* is the field name lowered
labels	*.frm.fgl_name*	
Scale	*.frm.fgls_name*	is text that shows the scale value

4.3 Packages

`Tk`
`tooltip` (if not present tooltip is ignored)

4.4 Dictionary

Name	Key	Value	Note
`afterMap`	*widgetName*	*widgetName*	
`afterWidgetMap`	*widgetName*	*List of widgets fields*	
`buttonsMap`	*fieldName*	*parameters*	List:: caption callBack
`defaultsMap`	*fieldName*	*value*	Contains also hidden values
`conversionTypeMap`	*AcceptedType*	*WidgetType*	Type used internally
`defaultLengthsMap`	*Type*	*defaultLength*	
`Elem`	*FieldName*	*pointers*	Pointers to widgets in fields list
`fieldsTypeMap`	*FieldName*	*type*	
`controlsMap`	*FieldName*	`controls`	List of controls
`scaleMap`	*FieldName*	`decimals`	Decimal digit to show (*)
`wReference`	*pathName* *FieldName*	*FieldName* *pathName*	
`keyValueMap`	*WidgetNameValue*	*Value* to return	For Radio buttons and Combo box
`rdbMap`	*WidgetNameValue*	*pathName*	For Radio buttons
`requiredButtonMap`	*WidgetType*	**CMB** 100 **B** 0 **C** 1 **H** 0	To determine which buttons to insert; default (others widgets) is `200`
`QSMap`	*FieldName*	`Qualitative list`	
`widgets`	*WidgetName*	`value`	

* if negative the value is shown and returned divide by `pow(10,abs($decimals))`.

4.5 Lists and arrays

Name	Value	Note
`allCommands`	`commands`	Default at start
`fields`	Components of widget	
`numericFieldsList`	Numeric types	For JSON output
`widgets`	List of widgets fields value	

4.6 Name spaces

Commons

4.7 Object oriented Form generator

Invoke a method from within another method: `my methodName`.

5 History

0.0.7 October 2020	• New widget Date, U UN not modifiable fields • Correct error in validate/required • Hidden values are now returned
0.0.8 Janvier 2021	• Fixed error in numeric fields (0 not accepted)

6 Annexes

6.1 Introduction to regular expressions

A regular expression is a string of characters used to search, check, extract part of text in a text; it has a cryptic syntax and here there is a sketch with a few examples.
The regular expression can be prefixed by modifiers such as **`(?i)`** to ignore the case.
The expression is formed with the characters to search in the text and control characters, among the latter there is a `\` said *escape* used to introduce the control characters or categories of characters:

- **`\` escape character**, for special characters (for example asterisk) or categories of characters:
  - **`\w`** any alphabetical and numerical character, **`\W`** any non alphabetical and numerical character,
  - **`\s`** *white space* namely. tabulation, line feed, form feed, carriage return, and space,
  - **`\d`** any numeric digits, **`\D`** any non digit,
- **`.`** any character,
- **quantifiers**, they apply to the character(s) that precede:
  - **`*`** zero or more characters
  - **`+`** one or more characters
  - **`?`** zero or one character (means possibly)
  - **`{n}`, `{n,}`** and **`{n,m}`** respective exactly *n* characters, almost *n* characters and from *n* to *m* characters.

`(...)` what is between parentheses is memorized,
`?=pattern` checks if *pattern* exists,
`[a-z]` any letter from `a` to `z` included,
`[a|b]` `a` or `b`,
`\b` word boundary,
`$` (at the bottom),
`^` (at start).

6.1.1 Examples

☞ Remember to escape the backslash and brackets, for example:

```
puts [regexp "^\\s*$" " \n"]
puts [regexp "^\[+-\]?\\d+$" "-97"]
```

`^\s*$`	Empty set or white spaces
`aa+`	Find a sequence of two or more `a`, like `aa`, `aaa`,....
`.(jpg|jpeg)$`	Controls file type jpg or jpeg[2]
`^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]`	Control of mail address

`{2,4}$`	
`^\d+$`	Only integers
`^[-+]?\d{1,2}(\.\d{1,2})?``$`	**Numeric values** **`[-+]?`** the sign is possible **`\d{1,2}`** one or two digits **`(\.\d{1,2})?`** It is possible to have a decimal point followed by one or two digits
`[aAbBcCdDeEfF\d]{8}`	8 hexadecimal digits

[2] With `-nocase` switch.