CLAS-NOTE 95-025

Tcl/Tk Interface to FORTRAN Programs

Vladimir S. Serov

Institute of Theoretical and Experimental Physics
B.Cheremushkinskaya, 25, Moscow, 117259 Russia

December 8, 1995

Contents

1 Introduction .

1.1 Why Tcl ? What is it ?

Why Tcl? Is it just another shell language? How can it help you?

Tcl stands for Tool Command Language. Tcl is really two things: a scripting language, and an interpreter for that language that is designed to be easy to embed into your application. Tcl and its associated X windows toolkit, Tk, were designed and crafted by Prof. John Ousterhout of U.C. Berkeley. These packages can be picked up off the Internet (see below) and used in your application, even a commercial one. The interpreter has been ported from UNIX to DOS and Macintosh environments.

As a scripting language, Tcl is similar to other UNIX shell languages such as the Bourne Shell, the C Shell, the Korn Shell, and Perl. Shell programs let you execute other programs. They provide enough programmability (variables, control flow, procedures) that you can build up complex scripts that assemble) existing programs into a new tool tailored for your needs. Shells are wonderful for automating routine chores.

It is the ability to easily add a Tcl interpreter to your application that sets it apart from other shells. Tcl fills the role of an extension language that is used to configure and customize applications. There is no need to invent a command language for your new application, or struggle to provide some sort of user programmability for your tool. Instead, by adding a Tcl interpreter you are encouraged to structure your application as a set of primitive operations that can be composed by a script to best suit the needs of your users. It also allows programmatic control over your application by other programs, leading to suites of applications that work together well.

There are other choices for extension languages that include Scheme, Elisp, and Python. Your choice between them is partly a matter of taste. Tcl has simple constructs and looks somewhat like C. It is also easy to add new Tcl primitives by writing C procedures. By now there are a large number of Tcl commands that have been contributed by the Tcl community. So another reason to choose Tcl is because of what you can access from Tcl scripts "out-of-the-box". To me, this is more important than the details of the language.

The Tcl C library has clean a interface and is simple to use. The library implements the basic interpreter and a set of core scripting commands that implement variables, flow control, file I/O, and procedures. In addition, your application can define new Tcl commands. These commands are associated with a C or C++ procedure that your application provides. The result is that applications are split into a set of primitives written in a compiled language and exported as Tcl commands. A Tcl script is used to compose the primitives into the overall application. The script layer has access to shell-like capability to run other programs and access the file system, as well as call directly into the application by using the application-specific Tcl commands you define. In addition, from the C programming level, you can call Tcl scripts, set and query Tcl variables, and even trace the execution of the Tcl interpreter.

There are many Tcl extensions freely available on the net. Most extensions include a C library that provides some new functionality, and a Tcl interface to the library. Examples include socket access for network programming, database access, telephone control, MIDI controller access, and expect, which adds Tcl commands to control interactive programs.

The most notable extension is Tk, a toolkit for X windows. Tk defines Tcl commands

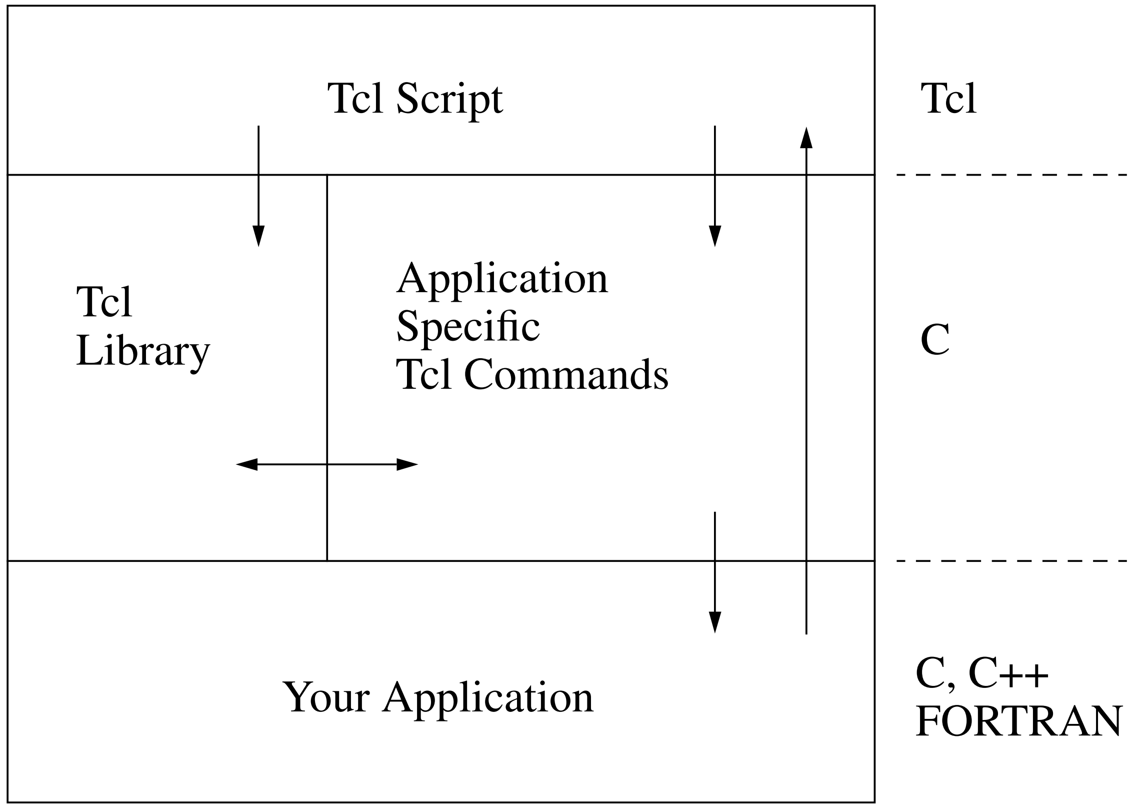

Figure 1: Structure of Application with Tcl interface

that let you create and manipulate user interface widgets. The script-based approach to UI programming has three benefits. First, development is fast because of the rapid turnaround - there is no waiting for long compilations. Second, the Tcl commands provide a higher-level interface to X than most standard C library toolkits. Simple interfaces require just a handful of commands to define them. At the same time, it is possible to refine the interface in order to get every detail just so. The fast turnaround aids the refinement process. The third advantage is that the user interface is clearly factored out from the rest of your application. The developer can concentrate on the implementation of the application core, and then fairly painlessly work up a user interface. The core set of Tk widgets is often sufficient for all your UI needs. However, it is also possible to write custom Tk widgets in C, and again there are many contributed Tk widgets available on the network.[1]

1.2 Organization of FORTRAN program using Tcl/Tk

The usual structure of a program uses Tcl as steering language is shown on Figure 1. Each Tcl command has an associated C function called every time the interpreter encounters this command in the script. The C function gets the arguments of Tcl the command and performs all desired actions. As it was mentioned, user can supply their own Tcl commands and associate with them C functions to implement specific actions relevant to his(her) application. Where is no difference between Tcl predefined commands and user supplied ones from the interpreter point of view.

[1]This is an introduction to Brent Welch book "Practical Programming in Tcl and Tk"

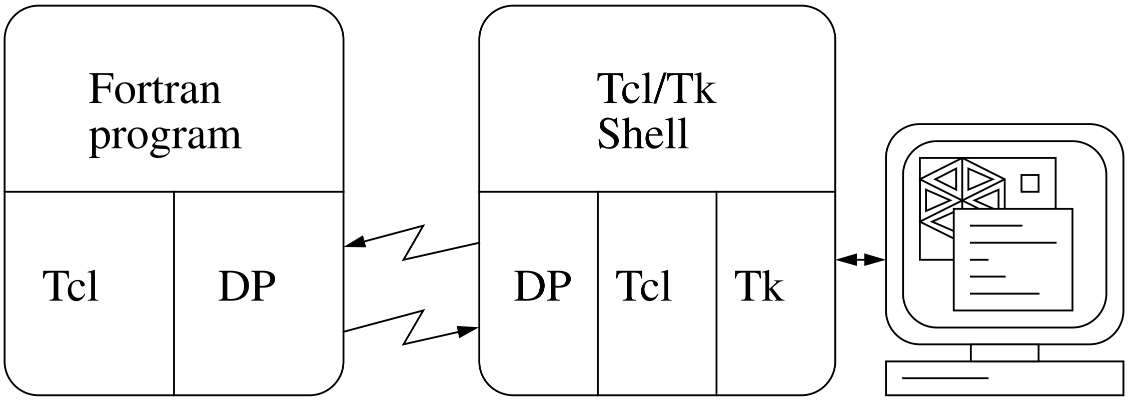

Figure 2: Structure of FORTRAN program with separated GUI

Commonly the development of a program which uses Tcl as steering language consist of following steps :

1. Splitting application into set of distinct primitive operations.
2. Making Tcl interface to such operation.
3. Writing Tcl convenience routines and final script to perform a whole work.
4. Writing GUI using Tk.
5. Possibly rewrite some time critical parts of Tcl script in C.

The Tk GUI requires some additional features. To be interactive, an X-windows interface must handle user actions at least several times a second. This fact imposes limitation on the length of calculation performed between user command processing. Such a limitation is very hard to meet for typical FORTRAN programs which perform massive calculations during generation and reconstruction of physical events. If embedded into a FORTRAN program the Tcl/Tk interface would be “dead” most the time. This problem could be solved by the separation of the FORTRAN program and the Tcl/Tk interface.

Communications are required if the interface and FORTRAN program are separated. An existing Tcl extension named Tcl-DP will handle this. DP stands for **D**istributed **P**rocessing. It allows to open network connections, send and receive data. Tcl-DP has the high level interface implementing RPC (**R**emote **P**rocedure **C**all) on the basis of the low level TCP network connection.

1.3 TkFor - Tcl extension for FORTRAN programs

TkFor - the Tcl extension which is intended for FORTRAN access from Tcl and vice versa. This extension was written in the process of developing a Tcl/Tk based shell for controlling physical simulation and reconstruction packages for the CLAS detector at CEBAF. TkFor supplies FORTRAN calls which create a Tcl interpretator embedded into a FORTRAN program. This interpretator has additional commands for accessing FORTRAN variables and packages. TkFor consist of two main parts :

- C and FORTRAN code for new Tcl commands, and for accessing Tcl from Fortran.
- Tcl scripts implementing Control Shell and communication between Shell and Fortran program.

TkFor features following :

- Tcl scripts executing from FORTRAN program.
- Tcl variable access from FORTRAN program.
- Tcl asynchronous events processing from FORTRAN program.
- FORTRAN common block access from Tcl.
- Tcl interface to FPACK package.
- Tcl interface to FFREAD package.
- Tcl interface to KUIP package.
- Tcl/Tk shell prototype for running and controlling FORTRAN programs that use TkFor package for communication with shell. The shell has the following features :
  - GUI to common blocks access.
  - GUI to FPACK package.
  - FORTRAN program Log file processing.

Typical FORTRAN programs for nuclear and particle physics flow from Initialization stage to the Main Loop with data processing and finally end with a Termination stage. There exists two ways to incorporate TkFor package into such FORTRAN program.

First one is presented in Figure 3. This way is easiest for starting with. It requires only a few subroutine calls at the Initialization stage and a call to the `TKYIELD` subroutine at the end of the event processing loop. The subroutine `TKYIELD` plays a key role. This routine provides an interface to the Tcl-DP event loop. Inside this loop pending RPC requests from the Shell program executed, which makes possible changes in the state of a FORTRAN program. For example, the user can use Tcl interface for common block access and interface to FFREAD package for changing the values of some variables inside a FORTRAN program. This can be done by sending RPC requests from Shell program through the network to the FORTRAN program. This request would be accepted asynchronously and executed when control flow in the FORTRAN program reaches the `TKYIELD` subroutine. The `TKYIELD` subroutine behavior is very flexible and controllable through the network, which brings wide capabilities for FORTRAN program steering. See chapter 2.4.1 for details.

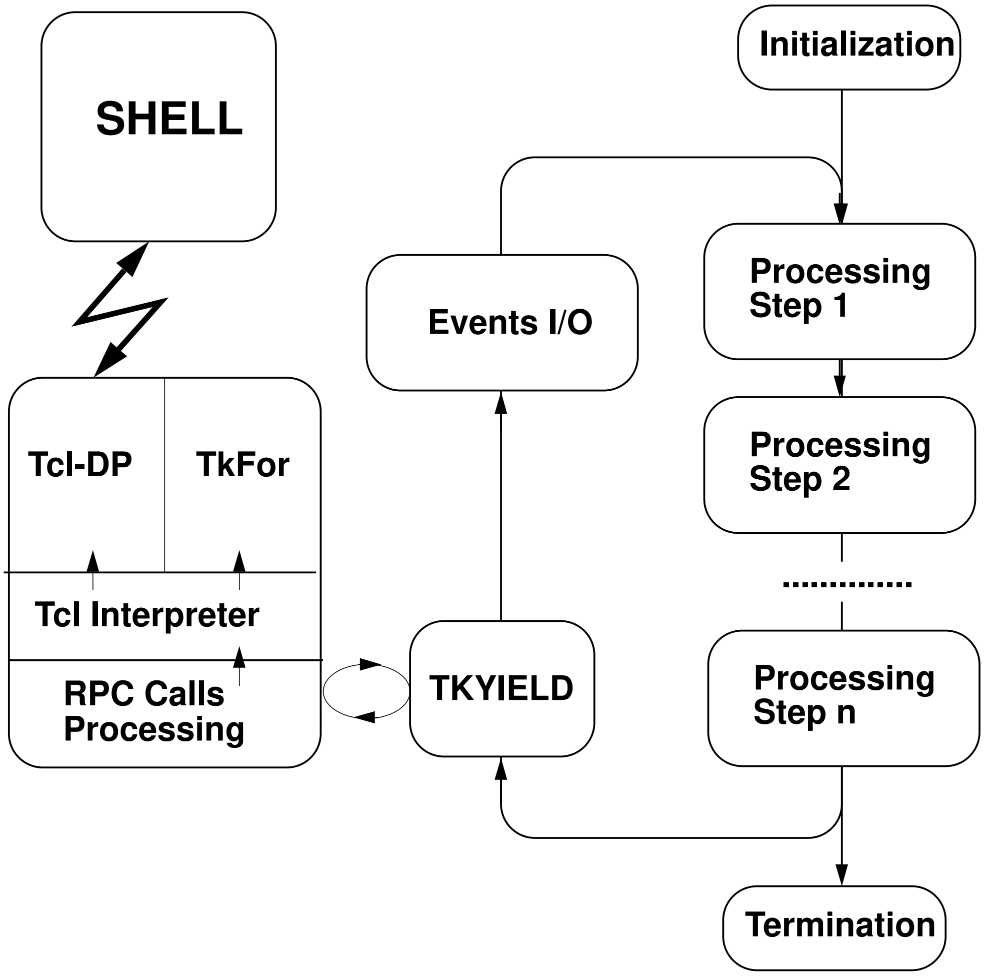

Figure 3: Structure of FORTRAN program with separated GUI

The econd way of Tcl and FORTRAN marriage displayed in Figure 4, is more natural and powerful but requires more work for implementation. This way is a twist of ordinary Tcl application structure shown on Figure 1. First, the user writes a Tcl interface to all actions in the FORTRAN program. Then, Tcl scripts are written to tie the functions together. This approach makes control flow much simpler and more controllable. More over, the user can customize program actions without long recompilations. It is still possible with this approach to control FORTRAN program through the network. The method of FORTRAN subroutine interfacing with Tcl commands is described in Appendix B.

Currently TkFor has only been tested on HP-UX 9.01 with Tcl 7.3 and Tk 3.6 .

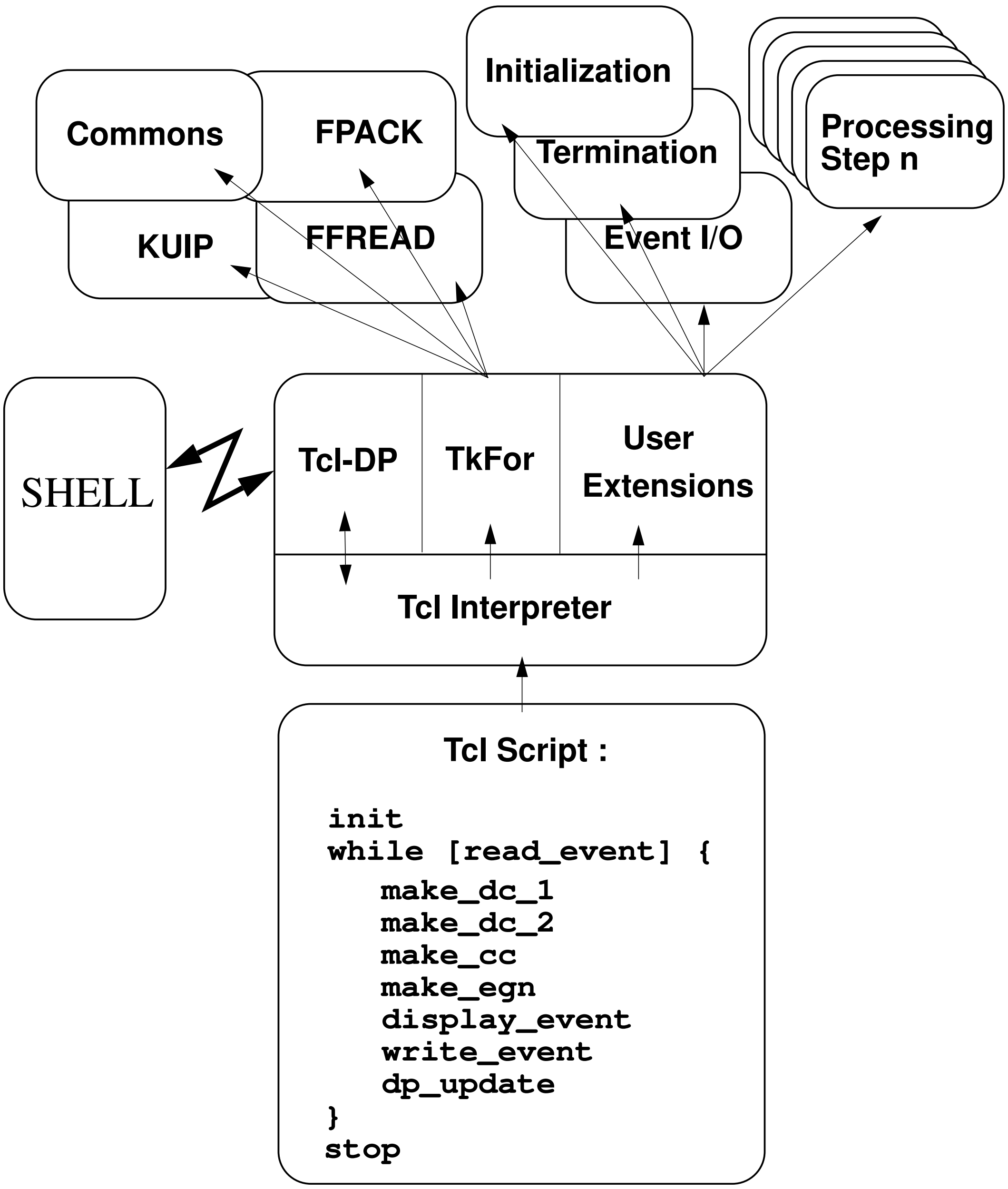

Figure 4: Structure of FORTRAN program with separated GUI

2 FORTRAN routines .

List of Fortran routines :

- TKINIT - Initialize full version of TkFor
- DPINIT - Initialize lite version of TkFor
- TKFOR_COMMONINIT - Initialize Tcl commands for common access
- TKFOR_FPACKINIT - Initialize Tcl commands for FPACK access
- TKFOR_FFREADINIT - Initialize Tcl commands for FFREAD access
- TKFOR_KUIPINIT - Initialize Tcl commands for KUIP access
- TCLEVAL - Evaluates Tcl command
- TCLFEVAL - Evaluates file with Tcl commands
- TCLRES - Extract result or error of last Tcl command
- TCLGETVAR - Extracts value of Tcl global variable
- TCLSETVAR - Sets value of Tcl global variable
- TKYIELD - Process Tcl asynchronous events
- TCL_INTERACTIVE - Prompts for user input
- AT_TKFOR_EXIT - Executed by Tcl at exit

2.1 Initialization .

You need to initialize the TkFor package before use. This is done by either a `TKINIT` call or `DPINIT` call. `TKINIT` starts an X11 version of TkFor, it's possible in this case to open windows from a FORTRAN program by evaluation of Tcl/Tk scripts. Though, it's possible to have X11 interface in FORTRAN, it's quite a difficult to make this interface interactive. So usage of lite version of TkFor is preferable. You can start such a version of TkFor by `DPINIT` call. Both versions of TkFor hava networking capabilities. Tcl-DP extension is used for network access, it provides an RPC capability to the program. See documentation on Tcl-DP for complete reference.

2.1.1 DPINIT

- Usage : `CALL DPINIT(`*STR,IERR*`)`
- Parameters :
  - *STR* - character string, if equals 'interactive' embedded Tcl interpretator will prompt you for Tcl commands, useful for debugging.
  - *IERR* - Integer variable , contains error code at exit. if 0 then `DPINIT` call was successful.
- Action : Initialize TkFor package without X11 interface (lite version)

2.1.2 TKINIT

- Usage : `CALL TKINIT(`*STR,IERR*`)`
- Parameters :
  - *STR* - character string, if equals 'interactive' embedded Tcl interpretator will prompt you for Tcl commands, useful for debugging.
  - *IERR* - Integer variable , contains error code at exit. if 0 then `TKINIT` call was successful.
- Action : Initialize TkFor package with X11 interface (full version)

If you want to use additional features of TkFor such as :

- FORTRAN common blocks access from Tcl
- Tcl interface to FPACK package
- Tcl interface to FFREAD package
- Tcl interface to KUIP package

You need to initialize these interfaces too. This is could be done by calling appropriate initialization subroutines :

- `CALL TKFOR_COMMONINIT(`*IERR*`)`
- `CALL TKFOR_FPACKINIT (`*IERR*`)`
- `CALL TKFOR_FFREADINIT(`*IERR*`)`
- `CALL TKFOR_KUIPINIT (`*IERR*`)`

All these subroutines set *IERR* to 0 in case of success.

WARNING !!! `TKFOR_FPACKINIT`, `TKFOR_FFREADINIT` and `TKFOR_KUIPINIT` do not initialize corresponding FORTRAN packages, so this work must be done by user.

At the initialization time the user must execute an initialization script. This script checks if the FORTRAN program was started from The Shell program or not and makes uppropriate arrangements. This is could be done including in a FORTRAN program the following statement :

```
CALL TCLEVAL('source $env(TKFOR)/TkForInit.tcl',IERR)
```

2.2 Tcl scripts execution from FORTRAN program .

It's possible to evaluate Tcl commands and files containing Tcl scripts from inside of a FORTRAN program. The `TCLEVAL` and `TCLFEVAL` subroutines are used for that respectively. One example of `TCLEVAL` usage is mentioned in previous paragraph. Details of usage.

2.2.1 TCLEVAL

- Usage : `CALL TCLEVAL(`*CMD,IERR)*
- Parameters :
  - *CMD* - character string containing Tcl command(s) to be evaluated.
  - *IERR* - Integer variable , contains error code at exit. if 0 then script execution was successful.
- Action : Evaluates *CMD* in root interpretator.

2.2.2 TCLFEVAL

- Usage : `CALL TCLFEVAL`*(FILE,IERR)*
- Parameters :
  - *FILE* - character string containing name of file to be evaluated.
  - *IERR* - Integer variable , contains error code at exit. if 0 then script execution was successful.
- Action : Evaluates *FILE* in root interpretator.

After Tcl command execution is defined the result of this command or error message, if this command fails. `TCLRES` subroutine is intended for such result or message extraction.

2.2.3 `TCLRES`

- Usage : `CALL TCLRES(`*STR*`)`
- Parameters : *STR* - character string, contains result or error message on exit.
- Action : result or error message of last Tcl command is returned as *STR* parameter value.

2.3 Tcl variables access from FORTRAN program .

It's possible to read and write Tcl variables from FORTRAN program - this is one of the ways for FORTRAN to communicate with Tcl and, further, with shell. `TCLGETVAR` and `TCLSETVAR` subroutines reads and writes Tcl global variables.

2.3.1 `TCLGETVAR`

- Usage : `CALL TCLGETVAR(`*VAR,VALUE,IERR*`)`
- Parameters :
  - *VAR* - character string, Tcl global variable name
  - *VALUE* - character string, contains extracted value on exit.
  - *IERR* - Integer variable , contains error code at exit. if 0 then reading was successful.
- Action : Read Tcl global variable

2.3.2 `TCLSETVAR`

- Usage :`CALL TCLSETVAR(`*VAR,VALUE,IERR*`)`
- Parameters :
  - *VAR* - character string, Tcl global variable name
  - *VALUE* - character string, it's value is copied to Tcl variable
  - *IERR* - Integer variable , contains error code at exit. if 0 then writing was successful.
- Action : Writes Tcl global variable

2.4 Tcl asynchronous event processing from a FORTRAN program

Tcl asynchronous event processing from a FORTRAN program is one of the main TkFor features. This possibility gives us a wide spectrum of remote control opportunities. FORTRAN program can do number crunching, while asynchronous events, in form of Tcl RPC calls are submitted to internal queue. Then, FORTRAN program can check this queue for pending events and process them. In case of full TkFor version not only network events can by processed but also an X11 ones. A `TKYIELD` subroutine performs these and some other functions. It's a good idea to use it at the end of the event loop and all other places where your FORTRAN program possibly could be controlled by the user. When your program is inside this call it can interact with you through the network and execute commands issued by the shell.

2.4.1 `TKYIELD`

- Usage : `CALL TKYIELD()`
- Parameters : none
- Action : affected by Tcl commands executed before or during this call. Tcl commands can be executed directly from FORTRAN or remotely by shell or user through the network.
  - Handle all pending RPC calls first and return if not other specified.
  - Begin to infinitely wait for other RPC calls if `'Suspend'` command was executed from FORTRAN before `'CALL TKYIELD()'` or during pending RPC call execution.
  - Can execute optional user script in the root interpreter during each call and if this script returns `'true'`, `'on'`, or 1 `TKIELD` begins waiting for RPC calls.
  - Finish waiting and return when receive command `'Resume'` or execute it as part of other script received from network.
  - Finish waiting and returns until next call when receive command `'OneStep'` or execute it as part of other script received from network.

Behavior of `TKYIELD` subroutine is controlled by four C variables :

- `int TkFor_StandAlone` - if equals to 1 that means there is no network connection so returns immediatle, if 0 - proceed futher. Installed to particular value during `TkForInit.tcl` script execution.
- `int TkFor_Wait_In_Yeild` - if equals to 1 `TKYIELD` doesn't return after processing all pending events but waits for new ones. This variable is set by `Suspend` and reset by `Resume` command
- `int TkFor_Fall_Through` - if equals to 1 `TKYIELD` sets this variable to 0 and returns. This variable is set by `OneStep` command.

- `char *TkFor_Wait_If` - if points to nonzero string `TKYIELD` treats this string as Tcl commands and executes it. If result of exeution is `'1'`, `'on'` or `'true'` `TKYIELD` sets `TkFor_Wait_In_Yeild` variable to 1 and starts waiting.

2.5 Making Tcl interactive .

It is possible to get Tcl prompt for commands using subroutine `TCL_INTERACTIVE`. After calling this subroutine Tcl interpreter will display the Tcl prompt : "`%`" ; read user commands from standart input and execute them. This subroutine will never return.

2.5.1 `TCL_INTERACTIVE`

- Usage : `CALL TCL_INTERACTIVE()`
- Parameters : none
- Action : Depends on user input.

2.6 User supplied routine executed before exiting .

User can supply subroutine named `AT_TKFOR_EXIT()` to perform all actions needed before exiting his program . A dummy version of this subroutine is placed into TkFor library to prevent unresolved references during linking. Subroutine `AT_TKFOR_EXIT()` is called from Tcl as "`exitfor`" command during execution of Tcl procedure "`RPCChildShutdown`". This procedure is executed by shell to stop FORTRAN program. Subroutine `AT_TKFOR_EXIT()` must not call `STOP` statement directly or indirectly.

3 Tcl interface for FORTRAN access .

All commands related to FORTRAN access exist only in interpreter embedded into FORTRAN program. These commands can be executed remotely by RPC calls by user or shell. To be available these commands must be initialized. Initialization procedure is described in "FORTRAN routines, Initialization" chapter.

3.1 FORTRAN common blocks access from Tcl .

After executing `CALL TKFOR_COMMONINIT(IERR)` three additional commands appears in the interpreter embedded into a FORTRAN program : `gfor`, `sfor` and `lforcommons`. `gfor` extracts value of FORTRAN variable in common block, `sfor` sets FORTRAN variable in common block and `lforcommons` returns a list commons that can be accessed.
Before describing syntacs of this commands let's remember memory model of FORTRAN common blocks. The main idea is that all variables in common block must be aligned on it's own length. This makes commons like :

```
REAL*4 R4
REAL*8 R8
COMMON /EXAMPLE1/ R4, R8
```

illegal. You will receive compiler warnings or errors about bad alignment. In case of warnings compiler just insert a 4 byte space between these variables to align `R8` on 8 byte boundary. This situation looks like every variable in common could be equivalenced with some element of array with same type and beginning from the common beginning. This fact was used as addressing model for `gfor` and `sfor` commands, they address common variables using type , length and index. For example, if you want to read or write `INTEGER*4` variable it can be placed in common on locations with byte offset from the beginning multiple to 4. That's like this variable is an element of `INTEGER*4` array and you only need to specify an index at this array to access all possible places where this variable could be. To determine such index for some particular variable in common you need count offset in bytes for this variable from the common beginning and divide it on the length of variable in bytes and after add 1. You must always got an integer number.

3.1.1 `gfor`

- Usage : `gfor` *common type length index*
- Parameters :
  - *common* - name of common you want to access (must be one of those that could be accessed, see `lforcommons` command for more information)
  - *type* - variable type : Int, Real, Bit or Char (case insensitive)
  - *length* - type length in bytes : 2 or 4 for Int, 4 or 8 for Real, 16 or 32 for Bit , from 1 to 1000 for Char.

- *index* - discussed before.

- Action : returns value of specified variable.

3.1.2 `sfor`

- Usage : `sfor` *common type length index value*
- Parameters :
  - *common* - name of common you want to access (must be one of those that could be accessed, see `lforcommons` command for more information)
  - *type* - variable type : Int, Real, Bit or Char (case insensitive)
  - *length* - type length in bytes : 2 or 4 for Int, 4 or 8 for Real, 16 or 32 for Bit , from 1 to 1000 for Char.
  - *index* - discussed before.
  - *value* - what to set variable to
- Action : sets value of specified variable.

3.1.3 `lforcommons`

- Usage : `lforcommons`
- Parameters : none
- Action : returns the list of common blocks that could be accessed. (at least `UCONTROL` is accessible)

Access to `UCONTROL` common block is predefined. To add access to other commons you need to run following command :

```
$TKFOR/mkcmn CMN_NAME1 CMN_NAME2 ... file_name.f
```

This command makes FORTRAN file specified as last argument with code needed for access to common blocks with names specified as other arguments in addition to `UCONTROL` common block. You need to link resulting code with your program.

3.2 Tcl interface to FPACK package .

After executing `CALL TKFOR_FPACKINIT(IERR)` an additional command appears in the interpreter embedded into FORTRAN program : `FPACK`. This command has two subcommands `eval` and `source`. `eval` is intended for evaluation of single `FPACK` command and `source` is intended for evaluation of file with `FPACK` commands.

3.2.1 `FPACK eval`

- Usage : `FPACK eval` *cmd*
- Parameters : *cmd* - FPACK command to evaluate
- Action : Executes subroutine `FPARM` with specified command. returns success or failure depending on FPACK error code.

3.2.2 `FPACK source`

- Usage : `FPACK source` *file*
- Parameters : *file* - name of file with `FPACK` commands to be evaluated.
- Action : Opens *file* using FPACK stream `TCLREADFILE` on first free unit and executes subroutine `FPARMR` with unit reported by FPACK for stream `TCLREADFILE`. Closes stream `TCLREADFILE`. Returns success or failure depending on FPACK error code.

3.3 Tcl interface to FFREAD package .

After executing `CALL TKFOR_FFREADINIT(IERR)` an additional command appears in the interpreter embedded into FORTRAN program : `FFREAD`. This command has six subcommands `info`, `source`, `eval`, `get` or `set`. `info` provides some information about FFREAD database, `eval` is intended for evaluation of single FFREAD command and `source` is intended for evaluation of file with FFREAD commands. `get` and `set` commands lets you directly read and write variables registered in FFREAD database.

3.3.1 `FFREAD info`

- Usage : `FFREAD info`
- Parameters : none
- Action : Dumps to stdout FFREAD database

3.3.2 `FFREAD info key`

- Usage : `FFREAD info key`
- Parameters : none
- Action : returns list of keys defined in FFREAD database

- Usage : `FFREAD info key` *pattern*
- Parameters : *pattern* - pattern for name of FFREAD keys you interested in.

- Action : returns information about each key matching the pattern in form of list. For each key specified: its name, type, length in number of variables and physical address of there beginning

3.3.3 `FFREAD info general`

- Usage : `FFREAD info general`
- Parameters : none
- Action : Returns a list of general characteristics of FFREAD database in form Keyword Value. Following keywords are defined : `NWORDS NCHKEY NWDKEY NKEYS MAXKEY NOFLEN NOFTYP NOFKEY LINPUT LUNIN LUNOUT ISTACK` .
  See FFREAD manual and code for description.

3.3.4 `FFREAD source`

- Usage : `FFREAD source` *file*
- Parameters : *file* - file with FFREAD commands to be evaluated.
- Action :Opens *file* using FPACK stream `TCLREADFILE` on first free unit Executes subroutine `FFGO` with unit reported by FPACK for stream `TCLREADFILE`. Closes stream `TCLREADFILE`. Returns success or failure depending on FPACK and FFREAD error codes.

3.3.5 `FFREAD eval`

- Usage : `FFREAD eval` *command*
- Parameters : *command* - FFREAD line
- Action : writes specified command into file `TCL_FFREAD.tmp` and executes after that Tcl command `FFREAD source TCL_FFREAD.tmp`

3.3.6 `FFREAD get`

- Usage : `FFREAD get` *key_name index [index] ...*
- Parameters :
  - *key_name* - FFREAD key
  - *index* - index of variable under this `key_name`
- Action : for each specified index returns value of variable with this index and key

3.3.7 `FFREAD set`

- Usage : `FFREAD set` *arg1 [arg2] ...*
- Parameters : each argument is list of {`KEY TYPE INDEX VALUE`}
- Action : for each argument sets variable of given `KEY TYPE` and `INDEX` to given `VALUE`

3.4 Tcl interface to KUIP package .

After executing `CALL TKFOR_KUIPINIT(IERR)` an additional command appears in the interpreter embedded into FORTRAN program : `KUIP`.

3.4.1 `KUIP`

- Usage : `KUIP` *command*
- Parameters : *command* - any valid KUIP command
- Action : Executes `ku_exel` function to evaluate KUIP commands in *command*. Returns success or failure depending on KUIP exit code.

4 TkFor setup .

Before linking and executing TkFor application it's necessary to properly setup a number of environment variables which are in use directly or indirectly by TkFor package. Presently, this job is automated by scripts on cebafb(h) cluster only. Following two lines is present in `.clascshrc` file do this job:

```
source ~serov/clas_tcl.setup
source ~serov/tkfor-0.3-setup
```

First of this lines installs CEBAF Computer center Tcl along with Tcl extensions in the CLAS public area. `clas_tcl.setup` file is the substitution for `setup tcl` command and you should use it instead of `setup tcl` command, not together. Second line sets environment variables related to the TkFor package itself. Here is list of them :

- `TKFOR` - points to the directory containing TkFor package, it's scripts, libraries , etc.
- `TKFOR_LITE_LIBS` - contains all necessary libraries for linking lite version of Tkfor; to be used in makefile for linking
- `TKFOR_FULL_LIBS` - contains all necessary libraries for linking full version of Tkfor; to be used in makefile for linking

Along with Tkfor `tkfor-0.2-setup` file installs attic and TIX packages also, so you need not setup them separately.

5 Compiling and linking .

There is no special compilation conditions for programs using TkFor package. The only one obligatory compiler option `+ppu` is due to the general CLAS convention to have FORTRAN object names with underscore appended. HP-UX FORTRAN 77 compiler doesn't append underscore by default, so we need to force it manually.
All libraries needed during linking program using TkFor are defined in environment variable `TKFOR_LITE_LIBS` or `TKFOR_FULL_LIBS`, depending on if you are using lite version of TkFor or full. Here is an example of linking the application that uses TkFor package:

```
fort77 tkfor.o -o tkfor $(TKFOR_LITE_LIBS) \
-L$(CLAS_LIB) -lfpack -lbos \
-L$(CERN_ROOT)/lib -lpacklib -lkernlib -lm
```

If you are not using Tcl interfaces to the FPACK, FFREAD and KUIP packages first line of above mentioned command is only one you need.

6 Appendix A – Example of FORTRAN program using Tcl/Tk

```
*
*       This demo represents the main features of TKFOR package
*       TKFOR - is a Tcl extension intended for communication
*       between Tcl and Fortran. Currently exist interface to :
*       - Fortran COMMON access ( call tkfor_commoninit() )
*       - FPACK  ( call tkfor_fpackinit()  )
*       - FFREAD ( call tkfor_ffreadinit() )
*       - KUIP   ( call tkfor_kuipinit()   )
*
*
*
      program main
*
*   --- Commons sizes
*
      parameter (ipaw_words=100000)
      parameter (ikuip_words=5000)
      parameter (iffread_words=1000)
      PARAMETER (NBCS=100000)

*
*   --- System Commons
*
        COMMON /BCS/      IW(NBCS)
      common /PAWC/     paw(ipaw_words)
      common /CFREAD/   space(iffread_words)
      common /UCONTROL/ itst(16)

*
*   --- variables
*
      character res*80, str*80, time*24, fdate*24, status*10
      external fdate

*
*   --- BOS init
*
      CALL BOS(IW,NBCS)

*
*   --- ZEBRA init
*
      call mzebra (-3)
      call mzpaw(ipaw_words,' ')
```

```
*
*   --- KUIP init
*
      call kuinit(ikuip_words)

*
*   --- FFREAD init
*
      call ffinit (iffread_words)
      call ffkey ('TSTI', itst(1),  5, 'INTEGER')
      call ffkey ('TSTR', itst(6),  5, 'REAL')
      call ffkey ('TSTL', itst(11), 5, 'LOGICAL')
      call ffkey ('TSTM', itst(1), 15, 'MIXED')

*
*   --- TKFOR-LITE init
*       It's recommended to use TKFOR-LITE which don't open it's own
*       windows because this can be done by shell controlling this
*       program
*
      call dpinit ('noninteractive',ierr)
      call tkfor_commoninit(ierr)
      call tkfor_fpackinit (ierr)
      call tkfor_ffreadinit(ierr)
      call tkfor_kuipinit  (ierr)
*
*   --- Run Tcl init script, it makes all arrangements needed later :
*       1. setup internal variables to default values
*       2. makes network connection to shell
*       3. dials back to shell and says that all O.K.
*
      call tcleval('source $env(TKFOR)/TkForInit.tcl',ierr)

*
*   --- Example of setting up Tcl variables from Fortran
*
      n=1000
      write (str,*) n
      call tclsetvar('delay',str,ierr)
      call tclsetvar('status','inactiv',ierr)
      call tclsetvar('x','0',ierr)
      call tclsetvar('time',fdate(),ierr)
*
*   --- Execute file with Tcl script
*
      call tclfeval ('t.tcl',ierr)
*
```

```
*   --- Example of extracting error code and result of last Tcl command
*
      call tclres (res)
      print *,'ERR:',ierr,'  RESULT:<',res(1:len_trim(res)),'>'

*
*   --- Suspend before entering events loop
*
      call tcleval('Suspend',ierr)
      call tkyield()

*
*   --- This is sample events loop
*
 10   continue
      i=i+1
      print *,' Event =',i

*
*   --- Example of extracting Tcl variables to Fortran
*
      call tclgetvar('delay',str,ierr)
      read (str,*,iostat=ioerr) n
      if (ioerr.ne.0) n=100000
*
*   --- Dummy calculations
*
      sum=0
      do j=1,n
         sum=sin(j*3.1416/j)+sum
      enddo

      write (str,*) i
      time=fdate()
      call tclsetvar('x',str,ierr)
      call tclsetvar('time',time,ierr)

*
*   --- Process pending events
*
      call tkyield()

      goto 10

      end
*
*
*
```

7 Appendix B – Interfacing fortran routine with Tcl command

This appendix doesn't pretend to be a complete explanation of how to write Tcl command which calls FORTRAN subroutines. Just an example of how to start writing an interface between Tcl and FORTRAN.

Writing new Tcl commands on C and Tcl library calls is described in detail in John Ousterhout's and Brent Welch's books on Tcl/Tk. The existing man pages on Tcl and TclX provide additional information. Unfortunaly Tcl command could not be written on FORTRAN, you need an intermediate C function which would be called by Tcl interpreter, parse arguments of Tcl command and call FORTRAN subroutine to perform action.

In the directory pointed by `TKFOR` environmental variable exists file `tkFor.h` which contains probably all necessary to start writing your own Tcl command. Just `#include <tkFor.h>` it in your C file describing new Tcl command.

There are some facts about C and FORTRAN calling conventions on the HP :

- On the most UNIX systems names of FORTRAN subroutines, functions, and commons are lowercase and have an underscore appended to the end. GNU f2c translator appends an additional underscore if it finds an underscore in the original name. Example :

  `SUBROUTINE TEST_INIT()` could be called from C as
  `test_init_();` if compiled with `'fort77 +ppu'` and as
  `test_init__();` if compiled with `'f2c'`.

- All arguments in FORTRAN is passed **by reference**. This means that the subroutine is passed pointer on the parameter location, so each parameter could by modified by the subroutine. Example :

  `SUBROUTINE TEST(IERR)` would have the following ANSI C prototype :
  `int test_(int *ierr);`
  and could be called from C as
  `int ierr;`
  `test_(&ierr);`

- On the most UNIX systems FORTRAN character strings arguments is presented by two arguments. First is pointer to the string beginning, second contains the length of the string. Second parameter is passed **by value** and appended after all explicit parameters. Example :

  `SUBROUTINE TEST2(I,CHR1,J,CHR2,K)` would have the following ANSI C prototype :
  ```
  int test2_(int* i, char* chr1, int* j, char* chr2, int* k,
             int chr1_lng, int chr2_lng);
  ```

here `chr1_lng` and `chr2_lng` represents length of passed strings `chr1` `chr2` respectively.

- Here variable types correspondence in FORTRAN and C for 32 bit computers :

  - `INTEGER*4 (INTEGER)        <--> int`
  - `INTEGER*2                  <--> short`
  - `LOGICAL*4 (LOGICAL)        <--> int`
  - `LOGICAL*2                  <--> short`
  - `REAL*4 (REAL)              <--> float`
  - `REAL*8 (DOUBLE PRECISION) <--> double`
  - `CHARACTER                  <--> char`

  The value of `.TRUE.` for FORTRAN logicals represented by `-1` ; the `.FALSE.` by `0`. In C this values is represented by `1` and `0` respectively.

For new Tcl command creation nesessary to write two C functions. The first of these functions would be called by Tcl interpreter when it encounter your command . The second one is link the name of your Tcl command with supplied C function which make the action. Second function must be called from FORTRAN at the initialization time.

Suppose your command has name `INIT` and intended for making some user initialization by calling following FORTRAN subroutine :

```
SUBROUTINE USER_INIT(STR,FACTOR,FLAG,NWORDS,IERR)
CHARACTER*(*) STR
REAL          FACTOR
LOGICAL       FLAG
INTEGER       NWORDS
INTEGER       IERR
```

The ANSI C prototype to this function would be following :

```
int user_init_(char* str, float* factor, int* flag, int* nwords, int* ierr,
               int str_lng);
```

Suppose we want to specify first four arguments as INIT command parameters and return IERR value as a result. Let's name the C action function `UserInit_Cmd` and second FORTRAN callable C function which will register Tcl command `INIT` as `user_tclcmd_init_`. To be called from FORTRAN as `CALL USER_TCLCMD_INIT(IERR)` this function must have following ANSI C prototype :

```
int user_tclcmd_init_(int* ierr);
```

Now put all this together in file `init_cmd.c` :

```
/*
 * File tkFor.h contains TkFor and Tcl defenitions
 */
#include <tkFor.h>
/*
 * Add our functions prototypes.
 *
 * First one is Action function, the set of it's parameters
 * is fixed by Tcl convention.
 *
 * Second would be called by FORTRAN as 'CALL USER_TCLCMD_INIT(IERR)'
 * and registers new Tcl command 'INIT'
 *
 * Third is prototype for user FORTRAN subroutine USER_INIT
 *
 */
int UserInit_Cmd (ClientData, Tcl_Interp *, int, char**);
int user_tclcmd_init_(int* ierr);
int user_init_(char* str, float* factor, int* flag, int* nwords, int* ierr,
               int str_lng);
/*
 * ================ C action routine ===================
 */
int UserInit_Cmd(clientdata, Interp, argc, argv)
     ClientData clientdata; /* Used only for Tk widgets            */
     Tcl_Interp *Interp;    /* Interpreter in which made action   */
     int         argc;      /* number of Tcl arguments             */
     char       *argv[];    /* Array of pointers on Tcl arguments */
{
  int          flag, nwords, ierr, str_lng;
  char         *str;
  double       dval;
  float        factor;

  if (argc != 5) {
/* produce error message if wrong number of arguments */
    Interp->result="Usage : INIT <str> <factor> <flag> <nwords>\n";
    return TCL_ERROR;
  }

  /* get first argument as character string .
   * copy contents of first argument in allocated space */
  str_lng = strlen(argv[1]);
  str     = strdup(argv[1]);
```

```
  /* get second argument as float number */
  ierr  = Tcl_GetDouble(Interp, argv[2], &dval);
  if (ierr!=TCL_OK) return TCL_ERROR;
  factor = (float)dval;

  /* get third argument as logical */
  ierr = Tcl_GetBoolean(Interp, argv[3], &flag);
  if (ierr!=TCL_OK) return TCL_ERROR;
  flag *= -1;

  /* get fourth argument as integer */
  ierr = Tcl_GetInt(Interp, listArgv[4], &nwords);
  if (ierr!=TCL_OK) return TCL_ERROR;

  /* call FORTRAN subroutine USER_INIT */
  user_init_(str, &factor, &flag, &nwords, &ierr, str_lng);

  /* free allocated space */
  free(str);

  /* return value of ierr as a result
   * so convert to ASCII string */
  strcpy(Interp->result, (char *)ltoa(ierr));

  return TCL_OK;
}

/*
 * ================ C registration  routine ===================
 */
int user_tclcmd_init_(int* ierr) {
  Tcl_CreateCommand(interp, 'INIT', UserInit_Cmd,
                    (ClientData)NULL, (Tcl_CmdDeleteProc *)NULL);
  *ierr=0;
}
```

Insert at your FORTRAN program following lines :

```
      CALL DPINIT('',IERR)
      CALL USER_TCLCMD_INIT(IERR)
      CALL TCL_INTERACTIVE
```

You would got a Tcl prompt and new Tcl command `INIT` would execute your subroutine

USER_INIT.

NOTE:
Of course you need to compile file init_cmd.c and link it with your program. This can be done by following commands :

```
cc -c -Aa -I$TCL_SRC/include -I$TKFOR init_cmd.c
fort77 -o myprog myprog.o init_cmd.o $TKFOR_LITE_LIBS mylib.a
```

8 Appendix C – DLL, a Tcl extension for dynamic library loading

8.1 Introduction

This Appendix contains documentation on the first release of dll - the Tcl extension which allows loading extensions of Tcl implemented as shared libraries. This is a modified and extended version of the dlwish extension written by Michael Moore (mdm@cis.ohio-state.edu). This DLL package supports multiple libraries during linking, it contains a means for managing multiple extensions. At this moment it has only been tested on HP-UX 9.01 with Tcl 7.3 and Tk 3.6 . This extension was written in a process of developing a Tcl/Tk based shell for controlling physical simulation and reconstruction packages for the CLAS detector in SURA/CEBAF. Presently source code installed on cebafb(h) claster in the `/clas/cebafb/u1/packages/tcltk/XW` directory, scripts resides in the `/clas/cebafb/u1/packages/tcltk/lib/dl` directory and executables in the `/clas/cebafb/u1/cmobj/tcltk/hpux.plt` .

8.2 Dynamic loading

The dynamic extension module takes advantage of dynamic loading of shared libraries under some OS's to load extensions in tcl/tk shells at run-time. The code provides a `DLL_Init()` function that creates a new Tcl command (load) that must be called in your `Tcl_AppInit()` function. Extensions to be loaded at run-time should be compiled as position independent code (option `+z` on HP-UX) and linked as shared libraries (option `-b` on HP-UX). They should define a public initialization function taking one argument - a pointer to the Tcl interpreter to be extended and returning `TCL_OK` or `TCL_ERROR` to report success or failure. Extensions complying with these rules can be loaded using the `'load'` Tcl command.

`'load'` command invocation : `load` *shared_libraries_list init_function_name* Where :

- *'shared_libraries_list'* - a Tcl list of libraries to be linked with full paths in a usual order to resolve external symbols.
- *'init_function_name'* - public initialization function to be invoked. This function must be present in the first library from the list.

Example :

```
load "$TCL_TK_SHLIBDIR/libtclx.sl $TCL_TK_SHLIBDIR/libtcl.sl \
 /lib/libm.sl /lib/libc.sl" TclX_Init
```

8.3 New shell building on HP-UX

To include new feature in your shell you need to include a call to `DLL_Init()` in your `Tcl_AppInit()` function. For example :

```
        if (DLL_Init(interp) == TCL_ERROR)  return TCL_ERROR;
```

Your need to link the resulting code with the `'platforms/hpux.c'` and also use `'-ldld'` option for dld library. For reference you can see `'Makefile'` and `'tcl*AppInit.c'` files.

In this directory is built shells with additional `'load'` command :

- `Xtcl` and `Xwish` - original tclsh and wish
- `Xtclx` and `Xwishx` - eXtended Tcl shells
- `Xwishxp` - eXtended Tcl shell with TkPixmap and TkIcon extensions
- `XStcl` and `XSwish` - Super shells with all extensions installed in CLAS common area.

The extended Tcl, TkPixmap and TkIcon extensions differ from other extensions in that they can't be loaded, but must be linked for proper performance, because they patches the original Tcl/Tk code.

8.4 Extension management facility

The DLL package provides scripts defining a set of routines for extension management. These scripts are automatically executed by the shell containing DLL extension at startup. Now scripts are in the directory `/clas/cebafb/u1/packages/tcltk/lib/dl`; the environment variable `DL_LIBRARY` must be pointing to this place for automatic sourcing of these scripts.

For extension loading you should provide some information about extension itself and it's loading procedure details. Command `'dll_RegisterExt'` serves this purpose. It stores in database the name of your extension, its version, required libraries, extensions, initialization procedure and some other information. After registering the extension you can load it using the `'dll_LoadExt'` command. This would be done according to information provided. The database keeps track of your actions and you can extract current status, what is registered, what is loaded, and all information about known extensions definition (see commands `dll_ReportExt dll_KnownExt dll_LoadedExt dll_IsKnown dll_IsLoaded`). By default debug information printing is turned OFF. Yo can turn it ON by the `'dll_Verbose'` command.

You can also use these commands for loading script only extensions. Just specify `{}` for list of libraries and initialization function of your extension. For loading and initialization in this case you can use `'init_script'` parameter at `'dll_RegisterExt'` call. Loading of script only extensions can be done even if shared libraries are not supported by the system or DLL extension wasn't linked.

At this moment, all extensions installed in the CLAS common area are described in this init script and available for loading into a shell. If you want to use your own extensions in this framework, you need to supply all needed information about them using the `'dll_RegisterExt'` command. Here are some tips on how to do that.

1. Be sure you are using a unique name for your extension or previous information will be replaced. You can extract names of already registered extensions using the `'dll_KnownExt'` command. WARNING ! Extensions names are case insensitive.
2. Most extensions can be loaded using only their own library, but if you get messages about unresolved symbols you need to specify additional libraries. Make sure you specify full path to the library(s).

3. It is better to do all initialization in the public initialization function provided in your extension. But some additional work can be done in the script specified for the 'init_script' parameter at the 'dll_RegisterExt' call. 'init_script' is also used for loading script only extensions.

4. If your extension requires some others, just specify the list of the names for the 'used_ext' parameter. This is the safest way to load all necessary extensions.

5. Checking if an extension is already present can be done by 'dll_IsLoaded' and 'dll_LoadedExt' commands if this extension was loaded using 'dll_LoadExt' command. But if it was linked it is usually much more difficult. If you want to prevent multiple loads of an extension you need to specify 'check_script' for checking the presence of extension. See 'dll_LoadExt' command description (step 4) for details. If You don't care about multiple loads specify {expr 0} - this returns 0, and means that extension would be loaded anyway.

6. See files Tcl_and_Tk_Ext.tcl, Tcl_Only_Ext.tcl and Tk_Only_Ext.tcl files in the /clas/cebafb/u1/packages/tcltk/lib/dl directory for examples.

So your first lines of Tcl/Tk script will probably looks like that :

```
-----------------------------------------------------------------------
#!/bin/sh
#\
exec $TCL_EXE/Xwishx -f "$0" ${1+"$@"}
# --- Tcl/Tk script begins here         ---
# --- Describe patched Tcl read command ---
set ext_name     newreadcmd
set ext_version  0.1
set lib_path     "$env(TKFOR)/libnewread.sl"
set init_fun     NewRead_Init
set init_script  {}
set used_ext     tcl
set export_cmd   {}
set check_script { expr 0 }; # -- this returns 0 : Load anyway
dll_RegisterExt $ext_name $ext_version $lib_path $init_fun $init_script \
    $used_ext $export_cmd $check_script
# --- Load needed extension
foreach ext {TclX TkX Tcl-DP expect itcl TIX tkfor newreadcmd} {
    if ![dll_LoadExt $ext] {
puts "===> Error !!! Can't load $ext extension"
# on error turn debugging on and try again to see diagnostics
dll_Verbose on
dll_LoadExt $ext
exit 1
    }
}
-----------------------------------------------------------------------
```

8.5 Breaf description of Tcl commands related with DLL

- `dll_RegisterExt` - submit all information about loadable extension into database
- `dll_ReportExt` - extract information about loadable extension from database
- `dll_LoadExt` - loads specified extension into shell
- `dll_KnownExt` - reports extensions names present in database
- `dll_LoadedExt` - reports names of loaded extensions
- `dll_IsKnown` - reports if specified extension is present in database
- `dll_IsLoaded` - reports if specified extension already loaded
- `dll_Verbose` - turn debug information printing on or off

8.5.1 `dll_RegisterExt`

- Invocation:
  `dll_RegisterExt` *ext_name ext_version lib_path init_fun* \
  init_script used_ext export_cmd check_script
- Parameters description :
  - *ext_name* - Name under which extension will be known
  - *ext_version* - Version
  - *lib_path* - Tcl list of shared libraries to be linked (see Dynamic loading)
  - *init_fun* - Function to be called for extension initialization (see Dynamic loading)
  - *init_script* - Tcl script to be executed for additional initialization
  - *used_ext* - List of extensions must be loaded before this one
  - *export_cmd* - List of commands exported by this extension
  - *check_script* - Tcl script checking if extension already exist
- Action : Creates (or replaces) information about extension in database and makes this extension loadable into shell
- Return : nothing .

See also `dll_LoadExt` for additional parameters description.

8.5.2 `dll_ReportExt`

- Invocation:
  `dll_ReportExt` *ext_name*

- Parameters description :
  ext_name - Name of extension you are interested about.

- Action : Extract information about extension from database in form of list

- Return : Tcl list with following elements :
  `ext_name ext_version lib_path init_fun init_script used_ext export_cmd check_script`

See also `dll_RegisterExt`.

8.5.3 `dll_LoadExt`

- Invocation:
  `dll_LoadExt` *ext_name*

- Parameters description :
  ext_name - Name of extension you need to load into shell.

- Action : Tries to load extension into shell in following steps :

  1. Check if Dynamic loading enabled (check for command `'load'` and global variable `'dl_version'`; if not found and you try to load extension that requires shared library return 0)
  2. Check if this extension is registered (only registered extensions can be loaded, so don't forget to do that using `'dll_RegisterExt'` command; if not registered return 0)
  3. Check if this extension is already loaded (if was loaded - return 1)
  4. Check if this extension was linked (`check_script` is executed and if this script return 1 this is mean that extension was linked and we return 1, else proceed to the step 5)
  5. Check for extensions must be previously loaded (try to load all needed extensions using `'dll_LoadExt'` command, if fail - return 0)
  6. Load shared libraries and execute `Init_Function` (attach shared libraries to current process and execute `Init_Function`, if specified, else do nothing; on error return 0)
  7. Execute init script (on error return 0, otherwise 1)

- Return : 1 if successful ; 0 otherwise

See also `dll_RegisterExt`.

8.5.4 `dll_KnownExt`

- Invocation: `dll_KnownExt`
- Parameters: no parameters
- Action : Reports extensions names present in database
- Return : List of extensions names registered by `dll_RegisterExt` command.

8.5.5 `dll_LoadedExt`

- Invocation: `dll_LoadedExt`
- Parameters : no parameters
- Action : Reports extensions names loaded into shell
- Return : List of loaded extensions names

8.5.6 `dll_IsLoaded`

- Invocation: `dll\_IsLoaded` *ext_name*
- Parameters description : *ext_name* - Name of extension you are interested about.
- Action : Checks if extension loaded into shell
- Return : 1 if loaded ; 0 otherwise

8.5.7 `dll_IsKnown`

- Invocation: `dll\_IsKnown` *ext_name*
- Parameters description : *ext_name* - Name of extension you are interested about.
- Action : Checks if extension is registered by `dll_RegisterExt` command.
- Return : 1 if registered ; 0 otherwise

8.5.8 `dll_Verbose`

- Invocation: `dll_Verbose` *[flag]*
- Parameters description : *flag* (optional) - logical value
- Action : if flag present verbose mode will be set ON or OFF. If flag equal `'1'`, `'on'`, `'true'` or `'yes'` verbose mode will be set ON, otherwise to OFF.
- Return : 1 if verbose mode is ON ; 0 otherwise

9 Appendix D – Tcl routines for background jobs management

9.1 Introduction

Tcl-DP provides Tcl commands for Tcl events loop access. Among them is `dp_whenidle` command which register Tcl command to be executed when program will be idle. This feature makes possible to collect data from installation asynchronously, and respond quickly on user input which has maximum priority. If you has a Tcl interface to data acquisition subroutines you can make data collection and processing while be prompted for commands input. TkFor package provides Tcl scripts for intuitive background job management. Tcl command `bg` can be used for describing background jobs, scheduling them and removing.

9.2 Tcl command `bg`

9.2.1 `bg create`

- Usage : `bg create` *period script*
- Arguments :
  - *period* is time in milliseconds between script invocation.
  - *script* is Tcl commands to be executed in background.
- Action : register a background job.
- Result : `id` for this job

9.2.2 `bg start`

- Usage : `bg start` *id*
- Arguments :
  - *id* is job identifier returned by `bg create` command
- Action : submit background job with *id* identifier for execution
- Result : id

9.2.3 `bg stop`

- Usage : `bg stop` *id*
- Arguments :

  - *id* is job identifier returned by `bg create` command
- Action : stop execution of background job with *id* identifier
- Result : id

9.2.4 `bg delete`

- Usage : `bg delete` *id*
- Arguments :

  - *id* is job identifier returned by `bg create` command
- Action : unregister background job with *id* identifier
- Result : id