Tcl_Eval Example Code

Here is some example code that uses Tcl_Eval for calling the user's Tcl code:

```
... lines and line_count come from our SVRF compiler.
... we use tvf_index to keep track of which TVF_FUNCTION block we
... are talking about.  tcl_proc_name is the name of the function we
.... we are calling to do evaluation inside the device script.

// Setup Code
//
// Read in the Tvf source file into a Tcl_DString buffer.
// This is done up front before any other processing.
//
tcl_scripts[tvf_index] = new TclDString;
Tcl_DStringInit( tcl_scripts[tvf_index] );
for ( unsigned int i=0; i<line_count; ++i ) {
    Tcl_DStringAppend( tcl_scripts[tvf_index], lines[i].line, -1 );
}

// Now set up a string that we can use to call the proc we loaded above
Tcl_DString call_string;
Tcl_DStringInit( &call_string );
Tcl_DStringAppend( &call_string, tcl_proc_name, -1 );
Tcl_DStringAppend( &call_string, " ", -1 );
// Add any necessary argument strings
for ( unsigned int i=0; i<arg_count; ++i ) {
    Tcl_DStringAppend( &call_string, tcl_arg[i], -1 );
    Tcl_DStringAppend( &call_string, " ", -1 );
}

// Create an interpreter
// This is done right before we start the device recognizer.
// We create one interpreter per execution thread.  We also seed it
// with a copy of the Tcl Proc that we are going to execute.
Tcl_Interp *interp = Tcl_CreateInterp();

//
//
int rc = Tcl_Eval(
       interp,
       Tcl_DStringValue(tcl_scripts[tvf_index])
);

...

// Calling Code
// Now, inside the device recognizer, we use another call to Tcl_Eval
// to call the proc that we stored earlier.  This time, we are
// passing device specific data as parameters to the call (that is all
// inside the Tcl_DString call_string.

int rc = Tcl_Eval(
        interp,
        Tcl_DStringValue(call_string)
);

// Various bookkeeping issues like checking for errors, destroying
// the strings and the interpreter have been left out for brevity.
```

Tcl_EvalEx Example Code

These calls take the place of the Tcl_Eval calls above

```
// In the setup code
int rc = Tcl_EvalEx(
    interp,
    Tcl_DStringValue(tcl_scripts[tvf_index]),
    -1,
    TCL_EVAL_GLOBAL
);

// and in the calling code
int rc = Tcl_EvalEx(
    interp,
    Tcl_DStringValue(call_string),
    -1,
    TCL_EVAL_GLOBAL
);
```

Tcl_EvalObjv Example Code

Instead of storing the strings in Tcl_DStrings, we will store them as an array of Tcl_Objs's that can keep track of the byte code as well:

```
// Setup code – we are also doing the setup in a Tcl_Obj.  This isn't
// really necessary since the interpreter is storing the Tcl proc we
// are calling internally anyway.
//
// Read in the Tvf source file into a Tcl_Obj.
// This is done up front before any other processing.
//
// This time, tcl_scripts is a Tcl_Obj** instead of a Tcl_DString**
//
tcl_scripts[tvf_index] = Tcl_NewObj();
Tcl_IncrRefCount( tcl_scripts[tvf_index] );
for ( unsigned int i=0; i<line_count; ++i ) {
    Tcl_AppendStringToObj( tcl_scripts[tvf_index], lines[i].line, NULL
);
}

// Now set up a string that we can use to call the proc we loaded above
Tcl_Obj** call_string = new Tcl_Obj*[arg_count+1];
call_string[0] = Tcl_NewStringObj( tcl_proc_name, -1 );
Tcl_IncrRefCount( call_string[0] );

// Add any necessary argument strings, also as Tcl_Obj's
for ( unsigned int i=0; i<arg_count; ++i ) {
    call_string[i+1] = Tcl_NewStringObj( tcl_arg[i], -1 );
    Tcl_IncrRefCount( call_string[i+1] );
}

// Create an interpreter
// This is done right before we start the device recognizer.
// We create one interpreter per execution thread.  We also seed it
// with a copy of the Tcl Proc that we are going to execute.
Tcl_Interp *interp = Tcl_CreateInterp();

//
```

```
//
int rc = Tcl_EvalObjEx(
       interp,
       tcl_scripts[tvf_index],
       TCL_EVAL_GLOBAL
);

// Calling code – This is the important part – we want to avoid byte
// compiling the call to our stored proc on each invocation.
//
// Now, inside the device recognizer, we use a call to Tcl_EvalObjv
// to call the proc that we stored there earlier.  This time, we are
// passing device specific data as parameters to the call (they are all
// inside the Tcl_Obj** call_string array along with the proc name.

int rc = Tcl_EvalObjv(
        interp,
        arg_count+1,
        call_string,
        TCL_EVAL_GLOBAL
);
```

Tcl_CreateObjCommand Example Code

```
// Set up code – for each argument, we create an ObjCommand object in
// in the global namespace of the interpreter.  These objects have names
// like arg_0_0 ( call number 0, argument number 0)
// This is done once upfront for each execution thread.  As successive
// devices are recognized, fields in the argument_data structure are
// are updated

for( int i=0; i<arg_count; ++i ) {
  char arg[ARG_BUF_SIZE];
  int len = sprintf( arg, "arg_%d_%d", call_idx, i );

  Ext_property_type type = work_type[call_site->arguments[i]];
  if ( type == EXT_PROPERTY_NUMBER ) {
      // this creates an object that represents a simple number
      Tcl_CreateObjCommand(
                        interpreter,
                        arg,
                        tcl_command_numeric_property, // Function
                        &call_info[call_idx].argument_data[i], // data
                        NULL
                  );
  } else if ( type == EXT_PROPERTY_ARRAY_INDEX ) {
      // This creates an object that represents one of our special
      // measurement arrays.
      Tcl_CreateObjCommand(
                        interpreter,
                        arg,
                        tcl_command_array_property, // Function
                        &call_info[call_idx].argument_data[i], // data
                        NULL
                  );
  }
  // Now create an argument object that will be part of the object array
  // that is passed to Tcl_EvalObjv for each device
  Tcl_Obj *arg_object = Tcl_NewStringObj(arg, len);
  Tcl_IncrRefCount(arg_object);
  call_info[call_idx].call_string[i+1] = arg_object;
}
```

Finally, we return to Tcl_EvalObjv as we saw above. This is the function that is called millions of times – once for each Tcl call in our script for each device in the design.

In order for the pre-created object to see the right data, we set a pointer that contains information about the current device for each object command.

```
for ( int i=0; i<arg_count; ++i ) {
     // initialize call specific data pointer
     // argument_data[i] was passed to Tcl_CreateObjCommand above.
     tvf_call.argument_data[i].device_info = device_info;
}

int rc = Tcl_EvalObjv(
        interp,
        arg_count+1, // # of arguments + 1 for the proc
        call_string, // This contains the proc and the arguments
        TCL_EVAL_GLOBAL
);
```

Tcl_SetObjResult Example Code

Here we use Tcl_SetObjResult to pass results back to the C++ program:

This code lives inside the tcl_command_array_property function that we used to create our Object command:

```
int tcl_command_array_property(
    ClientData cd,
    struct Tcl_Interp *interp,
    int objc,
    Tcl_Obj *CONST objv[]
)
{

  Ext_TVF_argument* arg_info = static_cast<Ext_TVF_argument*>(cd);

  const char* cmd = Tcl_GetStringFromObj( objv[1], NULL );

  // boring code that figures out what cmd is supposed to do...

  // Set the result and return
  if ( arg_info->result ) {
    Tcl_DecrRefCount(arg_info->result);
  }
  arg_info->result = Tcl_NewDoubleObj( d_result );
  Tcl_IncrRefCount( arg_info->result );
  Tcl_SetObjResult( interp, arg_info->result );
  return TCL_OK;

}
```

Here we try a caching scheme to avoid the back to back Tcl_DecrRefCount followed by Tcl_IncrRefCount when feasible. This didn't provide any performance advantage:

```
if ( !arg_info->result ) {
    // Create one if it wasn't there already
    arg_info->result = Tcl_NewDoubleObj( d_results );
    Tcl_IncrRefCount(arg_info->result);
else if ( Tcl_IsShared(arg_info->result) ) {
    // Don't use one that the script still needs
    Tcl_DecrRefCount(arg_info->result);
```

```
    arg_info->result = Tcl_NewDoubleObj( d );
    Tcl_IncrRefCount(arg_info->result);
} else
{
    // Use the currently unused result object again
    Tcl_SetDoubleObj(arg_info->result, d);
}
Tcl_SetObjResult( interp, arg_info->result );
return TCL_OK;
```

SVRF Example Code

```
// This is a device statement – it is Calibre's method for user defined
// intentional device recogntion
DEVICE MN NGATE NGATE SD SD <DIFF> <1>
[
   PROPERTY W, L, SEFFA, SEFFB, S
   S = ENCLOSURE_PERPENDICULAR( NGATE, DIFF, 1, 200 )

   // Calculations using builtin language exclusively
   W = SUM( S::W )
   L = AREA( NGATE ) / W
   SEFFA = W / SUM ( S::W / (S::A + 0.5*L ) ) - 0.5*L
   SEFFB = W / SUM ( S::W / (S::B + 0.5*L ) ) - 0.5*L
]

// This is the same device statement with SUM functionality replaced by
// TVF_NUMERIC_FUNCTION calls

DEVICE MN NGATE NGATE SD SD <DIFF> <1>
[
   PROPERTY W, L, SEFFA, SEFFB, S
   S = ENCLOSURE_PERPENDICULAR( NGATE, DIFF, 1, 200 )

   // Calculations using TVF Functions
   W = TVF_NUMERIC_FUNCTION( "sum_w", "device_func", S )
   L = AREA( NGATE ) / W

   SEFFA = TVF_NUMERIC_FUNCTION( "calc_eff_a", "device_func", S, W, L )
   SEFFB = TVF_NUMERIC_FUNCTION( "calc_eff_b", "device_func", S, W, L )
]

// Here is the Tcl implementation of the functions called above.

TVF FUNCTION device_func [/*
  proc sum_w { enc } {
     set w_acum 0.0
     set slice_count [ $enc slice_count ]
     for { set i 0 } { $i<$slice_count } { incr i } {
         set w_acum [ expr { $w_acum + [ $enc w $i ] } ]
     }
     return $w_acum
  }
  proc calc_eff_a { enc W L } {
     set acum 0.0

     set slice_count [ $enc slice_count ]
     for { set i 0 } { $i<$slice_count } { incr i } {
         set acum [ expr { $acum + [ $enc w $i ] / ( [ $enc a $i ] +
0.5*[$L] ) } ]
     }
     return [ expr { [$W] / $acum - 0.5 * [$L] } ]
  }
  proc calc_eff_b { enc W L } {
```

```
     set acum 0.0

     set slice_count [ $enc slice_count ]
     for { set i 0 } { $i<$slice_count } { incr i } {
          set acum [ expr { $acum + [ $enc w $i ] / ( [ $enc b $i ] +
0.5*[$L] ) } ]
     }
     return [ expr { [$W] / $acum - 0.5 * [$L] } ]
  }
*/]

// This version of the TVF FUNCTION used the experimental internal
// indexing functionality.
TVF FUNCTION device_func [/*
  proc sum_w { enc } {
     set w_acum 0.0
     $enc first_slice
     while { [ $enc remaining_slices ] > 0 }  {
         set w_acum [ expr { $w_acum + [ $enc w ] } ]
         $enc next_slice
     }
     return $w_acum
  }
  proc calc_eff_a { enc W L } {
     #puts "in calc_eff_a"
     set acum 0.0
     $enc first_slice
     while { [ $enc remaining_slices ] > 0 } {
         set acum [ expr { $acum + [ $enc w ] / ( [ $enc a ] + 0.5*[$L]
) } ]
         $enc next_slice
     }
     return [ expr { [$W] / $acum - 0.5 * [$L] } ]
  }
  proc calc_eff_b { enc W L } {
     #puts "in calc_eff_b"
     set acum 0.0
     $enc first_slice
     while { [ $enc remaining_slices ] > 0 } {
         set acum [ expr { $acum + [ $enc w ] / ( [ $enc b ] + 0.5*[$L]
) } ]
         $enc next_slice
     }
     return [ expr { [$W] / $acum - 0.5 * [$L] } ]
  }
*/]

// By removing the {} from expr commands, performance was slowed
// by 300%!
TVF FUNCTION device_func [/*
  proc sum_w { enc } {
     set w_acum 0.0
     set slice_count [ $enc slice_count ]
     for { set i 0 } { $i<$slice_count } { incr i } {
         set w_acum [ expr $w_acum + [ $enc w $i ] ]
     }
     return $w_acum
  }
  proc calc_eff_a { enc W L } {
     set acum 0.0

     set slice_count [ $enc slice_count ]
     for { set i 0 } { $i<$slice_count } { incr i } {
```

```
          set acum [ expr $acum + [ $enc w $i ] / ( [ $enc a $i ] +
0.5*[$L] ) ]
     }
     return [ expr [$W] / $acum - 0.5 * [$L] ]
  }
  proc calc_eff_b { enc W L } {
     set acum 0.0

     set slice_count [ $enc slice_count ]
     for { set i 0 } { $i<$slice_count } { incr i } {
          set acum [ expr $acum + [ $enc w $i ] / ( [ $enc b $i ] +
0.5*[$L] ) ]
     }
     return [ expr [$W] / $acum - 0.5 * [$L] ]
  }
*/]
```