XPS-Q8

Universal High-Performance Motion Controller/Driver

Newport®

Tcl Manual

V1.4.x

NOTE

Tcl/Tk **has been distributed freely for over 10 years and is now used in thousands of applications by companies and individuals worldwide. You are free to use it however you wish, even in commercial applications.**

Table of Contents

Universal High-Performance Motion Controller/Driver XPS-Q8

NOTE

This manual describes how to use Tcl scripts with the XPS-Q8 Controller. Several Tcl script examples have been provided to help illustrate key features of the XPS-Q8.

The Tcl drivers for the XPS-Q8 Controller have been developed for Tcl 8.4.19 interpreter and includes all XPS-Q8 functions.

For detailed Tcl documentation and support refer to the official tcl/tk tutorial: http://www.tcl.tk

1.0 TCP/IP Communication

XPS is based on a 10/100/1000 Base-T Ethernet communication link with TCP/IP protocol and uses a website approach for software tools and an FTP server for file transfers. This makes the XPS controller mostly independent from the user's operating system.. When networked, Unix, Linux or Windows users can access the same controller remotely from any place in the world. The full software feature set of the XPS is then available for code development, file transfer or general diagnostics.

The XPS firmware has been developed with a completely object-oriented approach. providing powerful, multi-parameter Function's (commands) in a more self-consistent and intuitive way than old-style mnemonic commands.

2.0 Tool Command Language

2.1 Introduction

Tcl is short for Tool Command Language. Tcl commands perform a variety of functions, such as: output a string, compute a math expression, or display a widget. Tcl casts everything into the mold of a command, even programming constructs like variable assignments and procedure definitions. Tcl requires a minimal amount of syntax to properly invoke commands, leaving the remainder of the work to command implementation.

The basic syntax of a Tcl command is:

```
command arg1 arg2 arg3 ...
```

The `command` is either the name of a built-in command or a Tcl procedure. White space (i.e., spaces or tabs) is used to separate the command name from command arguments, and a new line (i.e., the end of line character) or semicolon is used to terminate a command. Tcl does not interpret arguments respective to commands, other than to perform *`grouping`*. This allows multiple words to be used in a single argument. Tcl also performs *`substitution`*, which is used with programming variables and nested command calls. The behavior of the Tcl command processor can be summarized in three basic steps:

- Argument grouping.
- Value substitution of nested commands, variables, and backslash escapes.
- Command invocation. It is up to the command to interpret its arguments.

NOTE

The Tcl *gets* from stdio command is not supported.

2.2 Tcl scripting language features

2.2.1 "Hello, World!" example

```
puts stdout {Hello, World!}
```

⇒ *`Hello, World!`*

In this example, the command is **`puts`**, which takes two arguments: an I/O stream identifier and a string. **`puts`** writes the string to the I/O stream along with a trailing newline character.

There are two points to emphasize:

- Arguments are interpreted by the command. In the example, **`stdout`** is used to identify the standard output stream. The use of **`stdout`** as a name is a convention employed by **`puts`** and other I/O commands. Also, **`stderr`** is used to identify the standard error output, and **`stdin`** is used to identify the standard error input.
- Curly braces are used to group words together into a single argument. The **`puts`** command receives **`Hello, World!`** as its second argument.

`The braces are not part of the value.`

The braces are required syntax for the interpreter, and they get stripped off before the value is passed to the command. Braces group all characters, including newlines and nested braces, until a matching brace is found. Tcl also uses double quotes for grouping. Grouping arguments will be described in more detail later.

2.2.2 Variables

The **set** command is used to assign a value to a variable. It requires two arguments:

The first is the name of the variable, and the second is the value. Variable names can be any length, and case *is* recognized. In fact, you can use any character in a variable name.

NOTE

It is not necessary to declare Tcl variables before you use them.

The interpreter will create the variable when it is assigned a value.

The value of a variable is obtained later with the dollar-sign syntax, as illustrated in Example 1–2.

Example 1–2: Tcl variables.

```
set var 5
⇒ 5
set b $var
⇒ 5
```

The second **set** command assigns to variable **b** the value of variable **var**.

The use of the dollar sign is our first example of substitution. You might guess that the second **set** command substitutes the value of **var** for **$var** to obtain a new command.

```
set b 5
```

The actual implementation of substitution is more efficient, which is important when the value is large.

2.2.3 Command Substitution

The second form of substitution is ***command substitution***. A nested command is delimited by square brackets, **[ ]**. The Tcl interpreter takes everything between the brackets and evaluates it as a command. Rewriting the outer command by replacing the square brackets and everything between them with the result of the nested command. This is similar to the use of backquotes in other shells, except that it has the additional advantage of supporting arbitrary nesting of commands.

Example 1–3: Command substitution.

```
set len [string length foobar]
⇒ 6
```

In Example 1–3, the nested command is:

```
string length foobar
```

This command returns the length of the string **foobar**. The nested command runs first. Then, command substitution causes the outer command to be rewritten as if it were:

```
set len 6
```

If there are several cases of command substitution within a single command, the interpreter processes them from left to right. As each right bracket is encountered, the command it delimits is evaluated. This results in a sensible ordering in which nested commands are evaluated first so that their result can be passed as arguments to the outer command.

2.2.4 Math Expressions

The Tcl interpreter itself does not evaluate math expressions. Tcl just performs grouping, substitutions and command invocations. The **expr** command must be used to parse and evaluate math expressions.

Example 1–4: Simple arithmetic.

```
expr 7.2 / 4
```

⇒ *1.8*

The math syntax supported by **expr** is the same as the C expression syntax. The `expr` command deals with integer, floating point, and boolean values. Logical operations return either 0 (false) or 1 (true). Integer values are promoted to floating point values as needed. Octal values are indicated by a leading zero (e.g., `033` is `27` decimal). Hexadecimal values are indicated by a leading `0x`. Scientific notation for floating point numbers is also supported. A summary of the operator precedence is provided on page 20.

You can include variable references and nested commands in math expressions.

The following example uses **expr** to add the value of **x** to the length of the string **foobar**. As a result of the innermost command substitution, the **expr** command sees **6 + 7**, and **len** gets the value `13`:

Example 1–5: Nested commands.

```
set x 7
set len [expr [string length foobar] + $x]
```

⇒ *13*

The expression evaluator supports a number of built-in math functions. Example 1–6 computes the value of *pi*:

Example 1–6: Built-in math functions.

```
set pi [expr 2*asin(1.0)]
```

⇒ *3.1415926535897931*

The implementation of `expr` is careful to preserve accurate numeric values and avoid conversions between numbers and strings. However, you can make `expr` operate more efficiently by grouping the entire expression in curly braces. The explanation has to do with the byte code compiler that Tcl uses internally, the effects of which are explained in more detail on page 15. For now, you should be aware that these expressions are all valid and run a bit faster than the examples shown above:

Example 1–7: Grouping expressions with braces.

```
expr {7.2 / 4}
set len [expr {[string length foobar] + $x}]
set pi [expr {2*asin(1.0)}]
```

2.2.5 Backslash Substitution

The final type of substitution done by the Tcl interpreter is *backslash substitution*. This is used to quote characters that have special meaning to the interpreter. For example, you can specify a literal dollar sign, brace, or bracket by quoting it with a backslash. As a rule, however, if you find yourself using a lot of backslashes, there is probably a simpler way to achieve the effect you are striving

for. In particular, the `list` command will do quoting for you automatically. In Example 1–8 backslash is used to get a literal `$`:

Example 1–8: Quoting special characters with backslash.

```
set dollar \$foo
⇒ $foo
set x $dollar
⇒ $foo
Only a single round of interpretation is done.
```

The second `set` command in the example above illustrates an important property of Tcl. The value of `dollar` does not affect the substitution performed in the assignment to `x`. In other words, the Tcl parser does not care about the value of a variable when it does the substitution. In the example, the value of `x` and `dollar` is the string `$foo`. In general, you do not have to worry about the value of variables until you use `eval`.

You can also use backslash sequences to specify characters with their Unicode, hexadecimal, or octal value:

```
set escape \u001b
set escape \0x1b
set escape \033
```

The value of variable `escape` is the ASCII ESC character, which has character code `27`. The table on page 20 summarizes backslash substitutions.

A common use of backslashes is to continue long commands on multiple lines. This is necessary because a newline terminates a command. The backslash in the next example is required; otherwise the `expr` command gets terminated by the newline after the plus sign.

Example 1–9: Continuing long lines with backslashes.

```
set totalLength [expr [string length $one] + \
[string length $two]]
```

There are two fine points to consider when escaping newlines. First, if you are grouping an argument as described in the next section, then you do not need to escape newlines; the newlines are automatically a part of the group and will not terminate the command. Second, a backslash as the last character in a line is converted into a space, and all the white space at the beginning of the next line is replaced by this substitution. In other words, the backslash-newline sequence also consumes all the leading white space on the next line.

2.2.6 Grouping with Braces and Double Quotes

Double quotes and curly braces are used to group words together into one argument. The difference between double quotes and curly braces is that quotes allow substitutions to occur in the group, while curly braces prevent substitutions. This rule applies to command, variable, and backslash substitutions.

Example 1–10: Grouping with double quotes vs. braces.

```
set s Hello
⇒ Hello
puts stdout "The length of $s is [string length $s]."
⇒ The length of Hello is 5.
puts stdout {The length of $s is [string length $s].}
⇒ The length of $s is [string length $s].
```

In the second command of Example 1–10, the Tcl interpreter does variable and command substitution on the second argument to `puts`. In the third command, substitutions are prevented, so the string is printed as is. In practice, grouping with curly braces is used when substitutions on the argument must be delayed until a later time (or never at all). Examples include loops, conditional statements, and procedure

declarations. Double quotes are useful in simple cases like the `puts` command shown previously.

Another common use of quotes is with the `format` command. This is similar to the C `printf` function. The first argument to `format` is a format specifier, which often includes special characters like newlines, tabs, and spaces. The easiest way to specify such characters is with backslash sequences (e.g., `\n` for newline and `\t` for tab). The backslashes must be substituted before the `format` command is called, so you need to use quotes to group the format specifier.

```
puts [format "Item: %s\t%5.3f" $name $value]
```

Here `format` is used to align a name and a value with a tab. The `%s` and `%5.3f` indicate how the remaining arguments are to be formatted by the `format` command . Note that the trailing `\n` usually found in a C `printf` call is not needed because `puts` provides one for us.

2.2.6.1 Square Brackets Do Not Group

The square bracket syntax used for command substitution does not provide grouping. Instead, a nested command is considered part of the current group. In the command below, the double quotes group the last argument, and the nested command is just part of that group.

```
puts stdout "The length of $s is [string length $s]."
```

If an argument is made up of a nested command, you do not need to group it with double-quotes because the Tcl parser treats the whole nested command as part of the group.

```
puts stdout [string length $s]
```

The following is a redundant use of double quotes:

```
puts stdout "[expr $x + $y]"
```

2.2.6.2 Grouping before Substitution

The Tcl parser makes a single pass through a command as it makes grouping decisions and performs string substitutions. Grouping decisions are made before substitutions are performed, which is an important property of Tcl. This means the values being substituted will not affect grouping because the grouping decisions have already been made.

The following example demonstrates how nested command substitution affects grouping. A nested command is treated as an unbroken sequence of characters, regardless of internal structure. It is included with the surrounding group of characters when collecting arguments for the main command.

Example 1–11: Embedded command and variable substitution.

```
set x 7; set y 9
puts stdout $x+$y=[expr $x + $y]
```

⇒ *7+9=16*

In Example 1–11, the second argument to `puts` is:

```
$x+$y=[expr $x + $y]
```

The white space inside the nested command is ignored for the purposes of grouping the argument. By the time Tcl encounters the left bracket, it has already done some variable substitutions to obtain:

```
7+9=
```

When the left bracket is encountered, the interpreter calls itself recursively to evaluate the nested command. Again, the `$x` and `$y` are substituted before calling **`expr`**. Finally, the result of `expr` is substituted for everything from the left bracket to the right bracket. The **`puts`** command gets the following as the second argument:

```
7+9=16
```

Grouping before substitution.

The point of this example is to show how the grouping decision for **puts**'s second argument is made, before the command substitution is done. Even if the result of the nested command contained spaces or other special characters, they would be ignored for the purposes of grouping the arguments to the outer command. Grouping and variable substitution interact the same as with grouping and command substitution. Spaces or special characters in variable values do not affect grouping decisions because these decisions are made before the variable values are substituted.

If you want the output to look more readable as in the example, with spaces around the + and =, then you must use double quotes to explicitly group the argument to `puts`:

```
puts stdout "$x + $y = [expr $x + $y]"
```

The double quotes are used for grouping in this case to allow the variable and command substitution on the argument for `puts`.

2.2.6.3 Grouping Math Expressions with Braces

It turns out that **expr** performs its own substitutions inside curly braces. This is explained in more detail on page 15. This effectively means that you can write commands as listed below and still have substitutions on the variables in the expression occur:

```
puts stdout "$x + $y = [expr {$x + $y}]"
```

2.2.6.4 More Substitution Examples

If you have several substitutions without white space between them, you can avoid grouping with quotes. The following command **concat** applied to variables `a`, `b`, and `c` will concatenate them:

```
set concat $a$b$c
```

Again, if you want to add spaces, you'll need to use quotes:

```
set concat "$a $b $c"
```

In general, you can place a bracketed command or variable reference anywhere. The following computes a command name:

```
[findCommand $x] arg arg
```

2.2.7 Procedures

Tcl uses the **proc** command to define procedures. Once defined, a Tcl procedure is used just like any of other built-in Tcl command. The basic syntax to define a procedure is:

```
proc name arglist body
```

The first argument is the name of the procedure being defined. The second argument is a list of parameters used by the procedure. The third argument is a *command body* that is one or more Tcl commands. The procedure name is case sensitive, and in fact it can contain any character. Procedure names and variable names do not conflict with each other. As a convention, this guide begins procedure names with uppercase letters and variable names with lowercase letters. Adopting good programming conventions is important as your Tcl scripts become larger.

Example 1–12: Defining a procedure.

```
proc Diag {a b} {
set c [expr sqrt($a * $a + $b * $b)]
return $c
}
puts "The diagonal of a 3, 4 right triangle is [Diag 3 4]"
```

⇒ *The diagonal of a 3, 4 right triangle is 5.0*

The **Diag** procedure defined in the above example computes the length of the hypotenuse of a right triangle given the lengths of the legs. The **sqrt** function is one

of many math functions supported by the **expr** command. The variable c is local to the procedure; it is defined only during execution of **Diag**. Variable scope is discussed further in Section 7. It is not really necessary to use the variable c in this example. The procedure can also be written as:

```
proc Diag {a b} {
return [expr sqrt($a * $a + $b * $b)]
}
```

The return command is used to return the result of the procedure. The return command is optional in this example because the Tcl interpreter returns the value of the last command in the body as the value of the procedure. So, the procedure could be reduced to:

```
proc Diag {a b} {
expr sqrt($a * $a + $b * $b)
}
```

Note the use of curly braces in the example. The curly brace at the end of the first line starts the third argument to **proc**, which is the command body. In this case, the Tcl interpreter sees the opening left brace, causing it to ignore newline characters and scan the text until a matching right brace is found. *Double quotes have the same property*. They group characters, including

newlines, until another double quote is found. The result of the grouping is such that the third argument to **proc** is a sequence of commands. When they are evaluated later, the embedded newlines will terminate each command.

The other crucial effect of placing the curly braces around the procedure body is to delay any substitutions in the body, until the procedure is called. For example, the variables a, b, and c are not defined until the procedure is called, so we do not want to do variable substitution at the time Diag is defined.

The **proc** command supports additional features such as having a variable number of arguments and default values for arguments.

2.2.8 A Factorial Example

To reinforce what we have learned thus far, see the example below which uses a **while** loop to compute the factorial function:

Example 1–13: A while loop to compute a factorial.

```
proc Factorial {x} {
  set i 1; set product 1
  while {$i <= $x} {
    set product [expr $product * $i]
    incr i
  }
  return $product
}
Factorial 10
```

⇒ *3628800*

The **semicolon** used on the second line is there to remind you that it is a command terminator, just like the newline character. The **while** loop is used to multiply all numbers from one to the value of x. The first argument to **while** is a boolean expression, and the second argument is the command body to be executed.

The same math expression evaluator used by the **expr** command is used by **while** to evaluate the boolean expression. There is no need to explicitly use the **expr** command in the first argument to **while**, even if you have a much more complex expression.

The loop body and the procedure body are grouped with curly braces in the same way. The opening curly brace must be on the same line as **proc** and **while**. If you want to

put opening curly braces on the line after a **while** or an **if** statement, you must escape the newline with a backslash:

```
while {$i < $x} \
{
  set product ...
}
```

Always group expressions and command bodies with curly braces.

Curly braces around the boolean expression are crucial because they delay variable substitution until the `while` command implementation tests the expression. The following is an example that will result in is an infinite loop:

```
set i 1; while $i<=10 {incr i}
```

The loop will run indefinitely.* The reason is that the Tcl interpreter will substitute for `$i` *before* **while** is called, so **while** gets a constant expression `1<=10` that will always be true. You can avoid these kinds of errors by adopting a consistent coding convention that groups expressions with curly braces:

```
set i 1; while {$i<=10} {incr i}
```

The **incr** command is used to increment the value of the loop variable `i`. This is a handy command that saves us from the longer command:

```
set i [expr $i + 1]
```

The **incr** command can take an additional argument, a positive or negative integer by which to change the value of the variable. In this form, it is possible to eliminate the loop variable `i` and just modify the parameter `x`. The loop body can be written like this:

```
while {$x > 1} {
  set product [expr $product * $x]
  incr x -1
}
```

Example 1–14 shows another factorial, this time using a recursive definition. A recursive function is one that calls itself to complete its work. Each recursive call decrements `x` by one, until the value of `x` is equal to one, and then the recursion stops.

Example 1–14: A recursive definition of factorial.

```
proc Factorial {x} {
  if {$x <= 1} {
    return 1
  } else {
    return [expr $x * [Factorial [expr $x - 1]]]
  }
}
```

2.2.9 More about Variables

The `set` command will return the value of a variable only if it is passed in single argument. It treats that argument as a variable name and returns the current value of the variable. The dollar-sign syntax used to get the value of a variable is really just an easy way to use the `set` command. Example 1–15 shows a clever way to circumvent this restriction by putting the name of one variable into another variable:

Example 1–15: Using `set` to return a variable value.

```
set var {the value of var}
⇒ the value of var
set name var
```

```
⇒ var
set name
⇒ var
set $name
⇒ the value of var
```

This is a somewhat tricky example. In the last command, `$name` gets substituted with `var`. Then, the **`set`** command returns the value of `var`, which is `the value of var`. Nested **`set`** commands provide another way to achieve a level of indirection. The last `set` command above can be written as follows:

```
set [set name]
⇒ the value of var
```

Using a variable to store the name of another variable may seem overly complex, however, there are situations when it is very useful. There is even a special command, **`upvar`**, that makes this sort of trick easier.

2.2.9.1 Funny Variable Names

The Tcl interpreter makes some assumptions about variable names that makes it easy to embed variable references into other strings. By default, it will assume variable names contain only letters, digits, and the underscore. The construct `$foo.o` represents a concatenation of the value of `foo` and the literal "`.o`".

If the variable reference is not delimited by punctuation or white space, then you can use curly braces to explicitly delimit the variable name (e.g., `${x}`). You can also use this to reference variables with funny characters in their name, although generally the use of funny variable names is not ideal. If you find yourself using funny variable names, or computing the names of variables, then you may want to use the **`upvar`** command.

Example 1–16: Embedded variable references.

```
set foo filename
set object $foo.o
⇒ filename.o
set a AAA
set b abc${a}def
⇒ abcAAAdef
set .o yuk!
set x ${.o}y
⇒ yuk!y
```

2.2.9.2 The unset Command

You can delete a variable with the **`unset`** command:

```
unset varName varName2 ...
```

Any number of variable names can be passed to the **`unset`** command. However, **`unset`** will raise an error if a variable is not already defined.

2.2.9.3 Using info exists to check whether a variable exists

The existence of a variable can be tested with the **info exists** command. For example, because **incr** requires that a variable exist, you might have to test for the existence of the variable first.

Example 1–17: Using info to determine if a variable exists.

```
if {![info exists foobar]} {
  set foobar 0
} else {
  incr foobar
}
```

2.2.10 More about Math Expressions

This section describes a few of the finer points about mathematics in Tcl scripts. In Tcl 7.6 and earlier versions math is not very efficient due to required conversions between strings and numbers. The **expr** command must convert arguments from strings to numbers. It then does computations with double precision floating point values. The result is formatted into a string that have, by default, 12 significant digits. This number can be changed by setting the `tcl_precision` variable to the number of significant digits desired. Typically seventeen digits of precision are enough to insure that no information is lost when converting back and forth between a string and an IEEE double precision number:

Example 1–18: Controlling precision with tcl_precision.

```
expr 1 / 3
⇒ 0
expr 1 / 3.0
⇒ 0.333333333333
set tcl_precision 17
⇒ 17
expr 1 / 3.0
# The trailing 1 is the IEEE rounding digit
⇒ 0.33333333333333331
```

In Tcl 8.0 and later versions, the overhead of conversions is eliminated in most cases by the built-in compiler. Nevertheless, Tcl was not designed to support math-intensive applications. There is support for string comparisons by **expr**, so you can test string values with **if** statements. You must use quotes so that **expr** knows that you are doing a string comparison:

```
if {$answer == "yes"} { ... }
```

However, the **string** `compare` and **string equal** commands are more reliable because **expr** may do conversions on strings that look like numbers. Expressions can include variable and command substitutions and still be grouped with curly braces. This is because an argument to **expr** is subject to two rounds of substitution: one by the Tcl interpreter, and a second by **expr** itself. Ordinarily this is not a problem because math values do not contain the characters that are special to the Tcl interpreter. The second round of substitutions is needed to support commands like **while** and **if** that use the expression evaluator internally.

Grouping expressions can make them run more efficiently.

You should always group expressions in curly braces and let **expr** take care of command and variable substitutions. Otherwise, your values may suffer extra conversions from numbers to strings and back to numbers. Not only is this process slow, but the conversions can loose precision in certain circumstances. For example, suppose `x` is computed from a math function:

```
set x [expr {sqrt(2.0)}]
```

At this point the value of `x` is a double-precision floating point value, just as you would expect. If you do this:

```
set two [expr $x * $x]
```

then you may or may not get 2.0 as the result! This is because Tcl will substitute `$x` and **`expr`** will concatenate all its arguments into one string, and then parse the expression again. In contrast, if you do this:

```
set two [expr {$x * $x}]
```

then **`expr`** will do the substitutions, and it will be careful to preserve the floating point value of `x`. The expression will be more accurate and run more efficiently because no string conversions will be done. The story behind Tcl values is described in more detail in Chapter 44 on C programming and Tcl.

2.2.11 Comments

Tcl uses the pound character, **`#`**, for comments. Unlike the convention used by many other languages, the **`#`** must occur at the beginning of a command. A **`#`** that occurs elsewhere is not treated as a comment. An easy trick to append a comment to the end of a command is to precede the **`#`** with a **semicolon** to terminate the previous command:

```
# Here are some parameters
set rate 7.0 ;# The interest rate
set months 60 ;# The loan term
```

One subtle effect to watch for is that a **backslash** effectively continues a comment line onto the next line of the script. In addition, a **semicolon** inside a comment is not significant. Only a newline terminates comments:

```
# Here is the start of a Tcl comment \
and some more of it; still in the comment
```

A surprising property of Tcl comments is that curly braces inside comments are still counted for the purposes of finding brace pairs. This inconvenient "feature" was probably intended to keep the original Tcl parser simple. However, this means that the following will not work as expected:

```
# if {boolean expression1} {
if {boolean expression2} {
  some commands
}
```

The previous sequence results in an extra left curly brace, and probably a complaint about a missing closed brace at the end of your script! A good technique to comment out large chunks of code is to put the code inside an `if` block that will never execute:

```
if {0} {
  unused code here
}
```

2.2.12 Substitution and Grouping Summary

The following rules summarize the fundamental actions of grouping and substitution that are performed by the Tcl interpreter before it invokes a command:

- Command arguments are separated by **white space**, unless arguments are grouped with curly braces or double quotes as described below.
- Grouping with curly braces, **`{ }`**, prevents substitutions. Braces nest. The interpreter includes all characters between the matching left and right brace in the group, including newlines, semicolons, and nested braces. The enclosing (i.e., outermost) braces are not included in the group's value.
- Grouping with double quotes, **`" "`**, allows substitutions. The interpreter groups everything until another double quote is found, including newlines and semicolons. The enclosing quotes are not included in the group of characters. A double-quote character can be included in the group by quoting it with a backslash, (e.g., `\"`).

- **Grouping decisions** are made before substitutions are performed, which means that the values of variables or command results do not affect grouping.
- A dollar sign, **$**, causes variable substitution. Variable names can be any length, and are sensitive to case used. If variable references are embedded into other strings, or if they include characters other than letters, digits, and the underscore, they can be distinguished with the `${varname}` syntax.
- Square brackets, **[ ]**, cause command substitution. Everything between the brackets is treated as a command, and everything including the brackets is replaced with the result of the command. Nesting is allowed.
- The backslash character, ****, is used to quote special characters. You can think of this as another form of substitution in which the backslash and the next character or group of characters is replaced with a new character.
- **Substitutions** can occur anywhere unless prevented by curly brace grouping. Part of a group can be a constant string, and other parts can be the result of substitutions. Even the command name can be affected by substitutions.
- A single round of substitutions is performed before a command is invoked. The result of a substitution is not interpreted a second time. This rule is important if you have a variable value or a command result that contains special characters such as spaces, dollar signs, square brackets, or braces. Because only a single round of substitution is done, you do not have to worry about special characters in values causing extra substitutions.

2.2.13 Fine Points

- A common error is to forget a space between arguments when grouping with braces or quotes. This is because white space is used as the separator, while the braces or quotes only provide grouping. If you forget the space, you will get syntax errors about unexpected characters after the closing brace or quote. The following is an error because of the missing space between `}` and `{`:

```
if {$x > 1}{puts "x = $x"}
```

- A double quote is only used for grouping when it comes after white space. This means you can include a double quote in the middle of a group without quoting it with a backslash. This requires that curly braces or white space delimit the group. Using this obscure feature is not recommended, but this is what it looks like:

```
set silly a"b
```

- When double quotes are used for grouping, the special effect of curly braces is turned off. Substitutions occur everywhere inside a group formed with double quotes.

```
set x xvalue
set y "foo {$x} bar"
⇒ foo {xvalue} bar
```

- When double quotes are used for grouping and a nested command is encountered, the nested command can use double quotes for grouping, also.

```
puts "results [format "%f %f" $x $y]"
```

- Spaces are *not* required around the square brackets used for command substitution. For the purposes of grouping, the interpreter considers everything between the square brackets as part of the current group. The following sets `x` to the concatenation of two command results because there is no space between `]` and `[`.

```
set x [cmd1][cmd2]
```

- Newlines and semicolons are ignored when grouping with braces or double quotes. They get included in the group of characters just like all the others. The following sets `x` to a string that contains newlines:

```
set x "This is line one.
```

This is line two.

This is line three."

- During command substitution, newlines and semicolons `are` significant as command terminators. If you have a long command that is nested in square brackets, put a backslash before the newline if you want to continue the command on another line.
- A dollar sign followed by something other than a letter, digit, underscore, or left parenthesis is treated as a literal dollar sign. The following sets `x` to the single character `$`.

```
set x $
```

2.3 Reference

2.3.1 Backslash Sequences

\a	Bell. (0x7)
\b	Backspace. (0x8)
\f	Form feed. (0xc)
\n	Newline. (0xa)
\r	Carriage return. (0xd)
\t	Tab. (0x9)
\v	Vertical tab. (0xb)
\<newline>	Replace the newline and the leading white space on the next line with a space.
\\	Backslash. (‘\’)
ooo	Octal specification of character code. 1, 2, or 3 digits.
\x*hh*	Hexadecimal specification of character code. 1 or 2 digits.
\u*hhhh*	Hexadecimal specification of a 16-bit Unicode character value. 4 hex digits.
c	Replaced with literal *c* if *c* is not one of the cases listed above. In particular, \$, \", \{, \}, \], and \[are used to obtain these characters.

2.3.2 Arithmetic Operators

- ~ !	Unary minus, bitwise NOT, logical NOT.
* / %	Multiply, divide, remainder.
+ -	Add, subtract.
<< >>	Left shift, right shift.
< > <= >=	Comparison: less, greater, less or equal, greater or equal.
== !=	Equal, not equal.
&	Bitwise AND.
^	Bitwise XOR.
|	Bitwise OR.
&&	Logical AND.
||	Logical OR.
x?*y*:*z*	If *x* then *y* else *z*.

3.0 Tcl Script Execution at Boot

A Tcl boot script is a program that starts automatically after the controller boot sequence. It is defined in the *system.ini* configuration file under the GENERAL section.

```
[GENERAL]
BootScriptFileName = testarg.tcl
BootScriptArguments = arg1, arg2, arg3, arg4, arg5
```

BootScriptFileName is the file name of the Tcl script. This file must be stored in the *..\Admin\Public\Scripts* folder of the XPS controller.

BootScriptArguments defines the list of arguments of the Tcl script. The separator between two arguments is the comma.

Example:

A Tcl boot script could for instance contain the initialization and home search of all motion groups. Once the controller finishes booting, the motion groups will automatically initialize and home.

4.0 Principle of a Tcl Script Redirection to a telnet Session

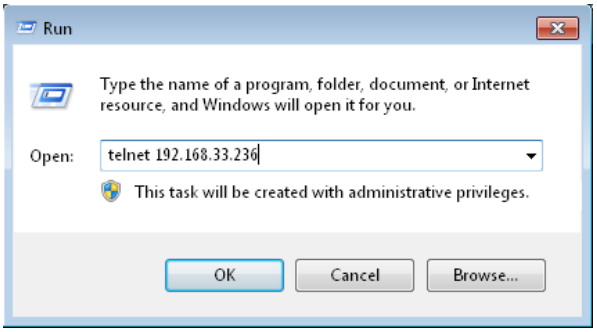
Opening a Telnet session is a convenient and easy way to observe Tcl Script responses, as well as pass information to and from the XPS controller.

4.1 Introduction

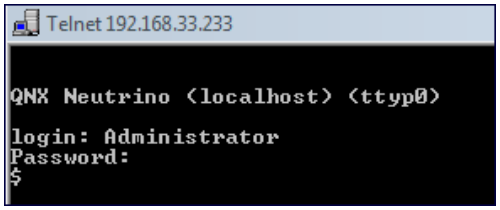
Telnet is a network protocol used on the Internet or local area networks to provide a bidirectional interactive text-oriented communications tool using a virtual terminal connection. User data is interspersed in-band with telnet control information in an 8-bit byte oriented data connection over the Transmission Control Protocol (TCP). This is a useful common protocol used to dialog with a remote machine.

A telnet connection is opened with any valid login, which can be administrator, anonymous, or whatever other logins are configured.

- For windows users, click *Start* -> *Run* -> then type *telnet* + IP address (IP address is the controller address you are connecting to) as below:



- The telnet window is opened, type login (here login and password are "Administrator"):



- An arrow appears which indicates that the telnet connection is ready for communication.
- Several telnet session window can be opened concurrently.

4.2 “Hello world !” Example

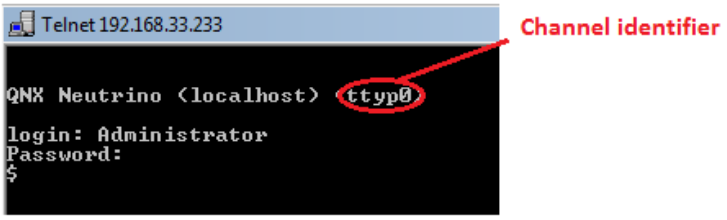
4.2.1 Tcl script example

The following Tcl script example shows how to display a message in a telnet window.

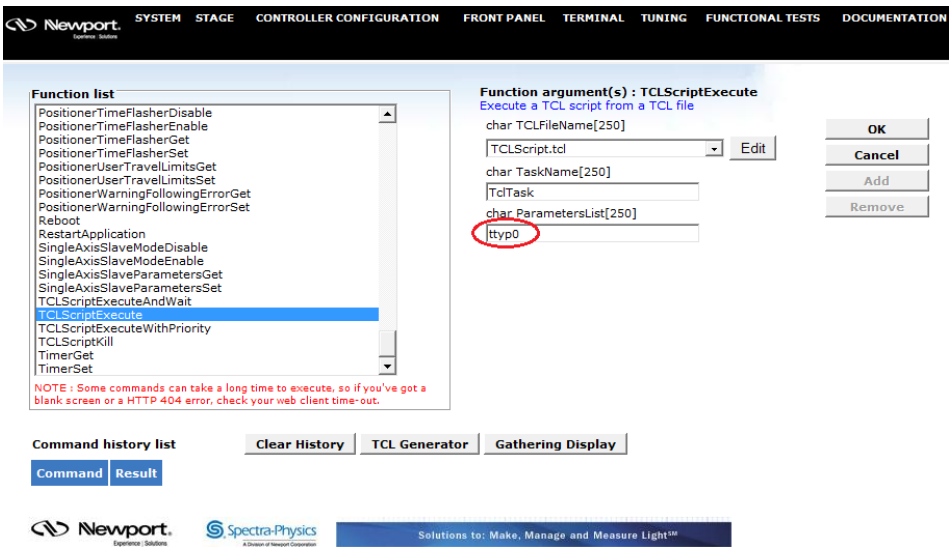
```
# Set channel’s name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}
# display hello world message
puts $telnetOut "Hello world !"

# Force transfer to channel’s output buffer
flush $telnetOut
```

4.2.2 Tcl script execution

When you open a telnet session you can see the channel’s identifier as shown below:

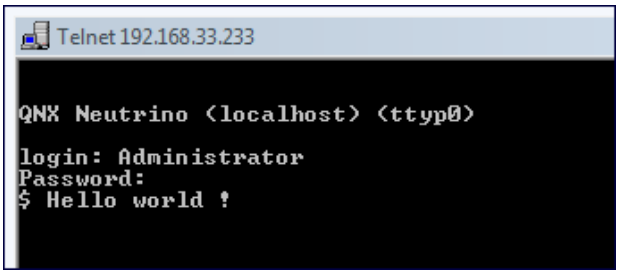


The channel identifier will be used as argument for the function called from Terminal to execute the Tcl script:



4.2.3 Tcl script execution result

The Tcl script execution result is shown on the opened telnet session window:



5.0 Example of a Tcl Script Redirection to a telnet Session

The following example shows the redirection of a Tcl script to a telnet session the telnet window displays the results of the Tcl execution (gets the library and the firmware version).

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}
# Get library version
set code [catch "GetLibraryVersion strVersion"]
if {$code != 0} {
ErrorStringGet $socketID $code strError
puts $telnetOut "GetLibraryVersion Not OK => error = $code : $strError"
# Force transfer to channel's output buffer
flush $telnetOut
} else {
puts $telnetOut "Library Version = $strVersion"
# Force transfer to channel's output buffer
flush $telnetOut
}
# Open socket
set TimeOut 60
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
puts $telnetOut "OpenConnection failed => $code"
# Force transfer to channel's output buffer
flush $telnetOut
} else {
# Get firmware version
set code [catch "FirmwareVersionGet $socketID strVersion"]
if {$code != 0} {
      ErrorStringGet $socketID $code strError
      puts $telnetOut "FirmwareVersionGet Not OK => error = $code :
$strError"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      puts $telnetOut "Firmware Version = $strVersion"
      # Force transfer to channel's output buffer
      flush $telnetOut
}
# Close TCP socket
set code [catch "TCP CloseSocket $socketID"]
}
```

6.0 Proposed Function for Error Handling

For convenient error debugging and safe program execution, the response (errors) of each XPS command should be read and tested. To do this, a procedure to “display error and close” can be used. This procedure is defined at the beginning of Tcl scripts. Users simply have to call this procedure after each API. This allows for a significant reduction of code when many APIs are used.

```
##########################################
#     Display error and close procedure   #
##########################################
proc DisplayErrorAndClose {socketID code APIName telnetOut} {

# Set global variable
global tcl argv

# If error occurred other than Timeout error
if {$code != -2} {
      # Error => Get error description
      set code2 [catch "ErrorStringGet $socketID $code strError"]

      # If error occurred with the API ErrorStringGet
      if {$code2 != 0} {
            # Display API name, error code and
            # ErrorStringGet error code
            # in the telnet window when using APIs
            # TCLScriptExecute or
            # TCLScriptExecuteAndWait
            puts $telnetOut "$APIName ERROR => $code /
            ErrorStringGet ERROR => $code2"

            # Force transfer to channel’s output buffer
            flush $telnetOut

            # in the web terminal when using API
            # TCLScriptExecuteAndWait
            set tcl argv(0) "$APIName ERROR => $code"
      } else {
            # Display API name, number and description
            # of the error
            # in the telnet window when using APIs
            #TCLScriptExecute or
            # TCLScriptExecuteAndWait
            puts $telnetOut "$APIName ERROR => $code :
            $strError

            # Force transfer to channel’s output
            # buffer
            flush $telnetOut

            # in the web terminal when using API
            # TCLScriptExecuteAndWait
            set tcl argv(0) "$APIName ERROR => $code : $strError"
      }
} else {
      # Display Timeout error
```

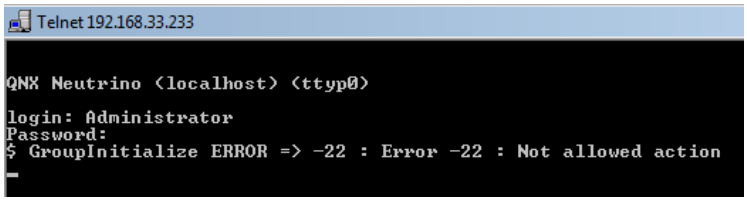
```
      # in the telnet window when using APIs
      # TCLScriptExecuteAndWait or
      # TCLScriptExecuteAndWait
      puts $telnetOut "$APIName ERROR => $code : TCP timeout"

      # Force transfer to channel's output buffer
      flush $telnetOut

      # in the web terminal when using API
      # TclScriptExecuteAndWait
      set tcl_argv(0) "$APIName ERROR => $code : TCP timeout"
}

      # Close TCP socket
      set code2 [catch "TCP_CloseSocket $socketID"]
      return
}
##########################################
#            Main process                #
##########################################
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Open socket
set TimeOut 60
set groupName "SingleAxis1"
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
puts $telnetOut "OpenConnection failed => $code"
# Force transfer to channel's output buffer
flush $telnetOut
} else {
       # Get firmware version
       set code [catch "GroupInitialize $socketID $groupName"]
       if {$code != 0} {
# Get error description
DisplayErrorAndClose $socketID $code "GroupInitialize"
       $telnetOut
  }

# Close TCP socket
set code [catch "TCP_CloseSocket $socketID"]
if {$code != 0} {
       puts $telnetOut "TCP socket $socketID failed => $code"
# Force transfer to channel's output buffer
flush $telnetOut
  }
```

This way of error management is also used with Tcl scripts that get generated by the Tcl generator, see Terminal of the XPS web interface. The procedure for displaying errors and closing the TCP connection is as described above. The DisplayErrorAndClose procedure can be used for each API in which the following code is used:

```
# Operation
set ErrorCode [catch "GroupInitialize $socketID S"]
# Error management
if {$ErrorCode != 0} {
    DisplayErrorAndClose $socketID $code "GroupInitialize"
    return
}
```

If an error occurs, the script returns the first error found, and indicates the API name that generated the error. Providing the error number as well as corresponding description. The execution of the script is stopped.

For instance, if we ask an initialized group to initialize again, the example code above returns the following error:



7.0 Examples of Tcl Programs

Please refer to the XPS Programmer's Manual for the list of XPS API's that are used with Tcl.

7.1 Using analog I/O for motion

7.1.1 Configuration

Group type	*Number*	*Group name*	*Positioner name*
XY	1	alignstation	alignstation.middle and alignstation.base

7.1.2 Description

This example opens a TCP connection, kills the XY group, then initializes and homes the group. Five relative moves of 1 unit each are commanded to the group. Then, the value of the GPIO2 analog input is read in a continuous loop and sent to the redirected *stdout* (refer to **Section 4**, ***Principle of a Tcl script redirection to a telnet session)*** as long as the voltage of the analog input is greater than 0.2 volt. While above 0.2 volts, absolute moves are commanded to both axes: the X positioner moves correspond to the voltage value of the analog input, and the Y positioner moves correspond to the opposite of the voltage value of the analog input. When the GPIO2 input voltage is equal to or less han the limit of 0.2 volt, the final move and display occurs. Finally the program ends by closing the socket.

If the voltage is less than 0.2 volt during the first reading, the program goes directly to the end without displaying the I/O value or absolute moves of the XY group.

Please see the **Section 6**, ***Proposed function for error handling*** for the code of the procedure ***DisplayErrorAndClose***.

7.1.3 Tcl code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 60
set group "alignstation"
set axis1 "alignstation.middle"
set axis2 "alignstation.base"
set analogin "GPIO2.ADC1"

# Open socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
puts $telnetOut "OpenConnection failed => $code"
# Force transfer to channel's output buffer
flush $telnetOut
```

```
} else {
# Kill group
set code [catch "GroupKill $socketID $group"]
if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupKill"  $telnetOut
      return
}

# Initialize group
set code [catch "GroupInitialize $socketID $group"]
if {$code != 0} {
DisplayErrorAndClose $socketID $code "GroupInitialize"  $telnetOut
return
}

# Home group
set code [catch "GroupHomeSearch $socketID $group"]
if {$code != 0} {
DisplayErrorAndClose $socketID $code "GroupHomeSearch"  $telnetOut
return
}

# Move group with 5 relative units
for { set var 0 } { $var <= 5 } { incr var } {
      set code [catch "GroupMoveRelative
                  $socketID $group 1 1"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code
      "GroupMoveRelative" $telnetOut
      return
      }
}

# Get analog value
set code [catch "GPIOAnalogGet $socketID $analogin voltage"]
if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GPIOAnalogGet" $telnetOut
      return
}

# Test if voltage is greater than 0.2 volt
while { $voltage >= 0.2 } {

      # Get analog value
      set code [catch "GPIOAnalogGet
                  $socketID $analogin voltage"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "GPIOAnalogGet" $telnetOut
            return
      }
      set move1 $voltage
      set move2 [expr { $voltage * -1 } ]
      puts $telnetOut "$analogin: $voltage volt(s)"
      # Force transfer to channel's output buffer
```

```
      flush $telnetOut
      puts $telnetOut "           move axis1: $move1"
      flush $telnetOut
      puts $telnetOut "           move axis2: $move2"
      flush $telnetOut

      # Move axis 1
      set code [catch "GroupMoveAbsolute $socketID
                  $axis1 $move1"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "GroupMoveAbsolute" $telnetOut
            return
      }
      # Move axis 2
      set code [catch "GroupMoveAbsolute $socketID
                  $axis2 $move2"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "GroupMoveAbsolute" $telnetOut
            return
      }
}

# Wait 1 second and close socket
after 1000
puts $telnetOut "End of program"

# Force transfer to channel's output buffer
flush $telnetOut

TCP CloseSocket $socketID
```

This is what gets displayed on a Telnet window. In this example the input voltage of GPIO2 decreases from 2.4 V to 0 V.

7.2 Using Digital I/O for Motion

7.2.1 Configuration

Group type	Number	Group name	Positioner name
XY	1	alignstation	alignstation.middle and alignstation.base

7.2.2 Description

This example opens a TCP connection, kills the XY group, then initializes and homes the group. Five relative moves of 1 unit each are commanded to the group. Then, the value of the GPIO1 digital input is read in a continuous loop and sent to the redirected *stdout* as long as the value of the input is not 255 (Refer to **Section 4**, ***Principle of a Tcl script redirection to a telnet session***). When the value of the digital GPIO1 input is equal to 1, absolute moves are commanded to both axes: the X positioner moves to the absolute position 1 and the Y positioner moves to the absolute position -1. When the value of 255 is obtained, the last display occurs. Finally, the program ends by closing the socket. If the GPIO1 input value is 255 during the first reading, the program goes to the end without displaying the digital input value.

Please see the **Section 6**, ***Proposed function for error handling***.

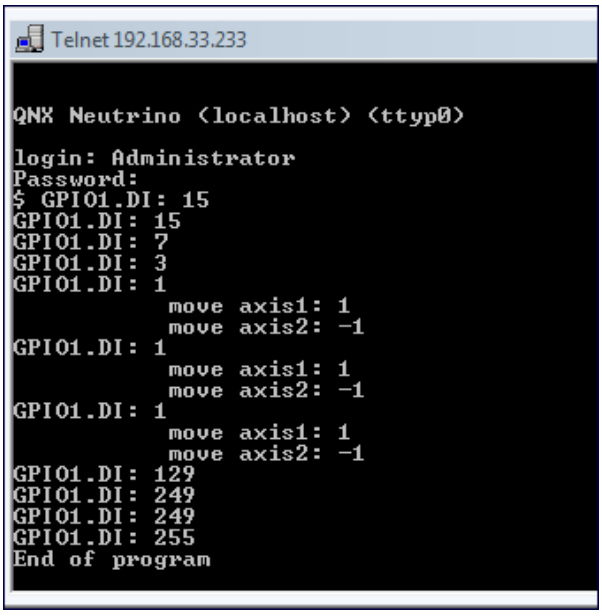
7.2.3 Tcl Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 60
set group "alignstation"
set axis1 "alignstation.middle"
set axis2 "alignstation.base"
set digitalin "GPIO1.DI"

# Open socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      # Kill group
      set code [catch "GroupKill $socketID $group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupKill" $telnetOut
            return
      }
```

```
      # Initialize group
      set code [catch "GroupInitialize $socketID $group"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupInitialize" $telnetOut
      return
}

# Home group
set code [catch "GroupHomeSearch $socketID $group"]
if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupHomeSearch" $telnetOut
      return
}

# Move group with 5 relative units
for { set var 0 } { $var <= 5 } { incr var } {
      set code [catch "GroupMoveRelative $socketID
                  $group 1 1”]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupMoveRelative"
$telnetOut
            return
      }
}

# Get digital value
set code [catch "GPIODigitalGet $socketID $digitalin value"]
if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GPIODigitalGet" $telnetOut
      return
}

# Test if value of GPIO1.DI is different from 255
while { $value != 255 } {

      # Get digital value
      set code [catch "GPIODigitalGet $socketID
                  digitalin value”]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "GPIODigitalGet" $telnetOut
            return
      }
      puts $telnetOut "$digitalin: $value"
      flush $telnetOut
      if { $value == 1 } {
            puts $telnetOut "           move axis1: 1"
            flush $telnetOut
            puts $telnetOut "           move axis2: -1"
            flush $telnetOut
            # Move axis 1
            set code [catch "GroupMoveAbsolute $socketID
                        $axis1 1"]
            if {$code != 0} {
```

```
                DisplayErrorAndClose $socketID $code
                "GroupMoveAbsolute" $telnetOut
                return
            }

            # Move axis 2
            set code [catch "GroupMoveAbsolute $socketID
                        $axis2 -1"]
            if {$code != 0} {
                DisplayErrorAndClose $socketID $code
                "GroupMoveAbsolute" $telnetOut
                return
            }
        } else {
            after 100
        }
        after 1000

}

# Wait 1 second and close socket
after 1000
puts $telnetOut "End of program"

# Force transfer to channel's output buffer
flush $telnetOut

TCP_CloseSocket $socketID

}
```

This is what gets displayed on the telnet window. See Section 4, ***Principle of a Tcl script redirection to a telnet session*** for details about telnet connections:



7.3 GPIO1 Test

7.3.1 Description

This example opens a TCP connection. It sets the value to 255 to the mask and the output GPIO1.DO, then gets this output value and assigns it to the variable OA. The program sets the value of 255 to the mask and the value of 0 to the output GPIO1.DO, then gets this output value and assigns it to variable OB. It sets the value of 63 to the mask and the value of 255 to the output GPIO1.DO, then gets this output value and assigns it to the variable OC. After the settings, the program tests the value of the variables OA, OB and OC. Finally, the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

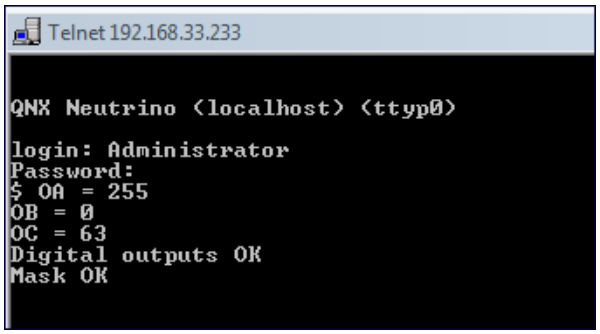
6 Proposed function for error handling.

7.3.2 Tcl Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 120
set output  "GPIO1.DO"
set input   "GPIO1.DI"
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
  puts $telnetOut "OpenConnection failed => $code"
  # Force transfer to channel's output buffer
  flush $telnetOut
} else {
  # Set output of GPIO1 to 255
  set code [catch "GPIODigitalSet $socketID $output 255 255"]
  if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalSet" $telnetOut
    return
  }
  # Get value of output of GPIO1 and store it in OA
  set code [catch "GPIODigitalGet $socketID $output OA"]
  if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GPIODigitalGet" $telnetOut
    return
  } else {
    puts $telnetOut "OA = $OA"
# Force transfer to channel's output buffer
flush $telnetOut
  }
```

```
# Set output of GPIO1 to 0
set code [catch "GPIODigitalSet $socketID $output 255 0"]
if {$code != 0} {
  DisplayErrorAndClose $socketID $code "GPIODigitalSet" $telnetOut
  return
}
# Get value of output of GPIO1 and store it in OB
set code [catch "GPIODigitalGet $socketID $output OB"]
if {$code != 0} {
  DisplayErrorAndClose $socketID $code "GPIODigitalGet" $telnetOut
  return
} else {
  puts $telnetOut "OB = $OB"
  # Force transfer to channel's output buffer
  flush $telnetOut
}
# Set output of GPIO1 to 63 (mask value)
set code [catch "GPIODigitalSet $socketID $output 63 255"]
if {$code != 0} {
  DisplayErrorAndClose $socketID $code "GPIODigitalSet" $telnetOut
  return
}
# Get value of output of GPIO1 and store it in OC
set code [catch "GPIODigitalGet $socketID $output OC"]
if {$code != 0} {
  DisplayErrorAndClose $socketID $code "GPIODigitalGet" $telnetOut
  return
} else {
  puts $telnetOut "OC = $OC"
  # Force transfer to channel's output buffer
  flush $telnetOut
}
# Test if OA = 255 and OB = 0
if {$OA == 255 & $OB == 0} {
  puts $telnetOut "Digital outputs OK"
  # Force transfer to channel's output buffer
  flush $telnetOut
  # Test if OC = 63
  if {$OC == 63} {
    puts $telnetOut "Mask OK"
    # Force transfer to channel's output buffer
    flush $telnetOut
  } else {
    puts $telnetOut "Problem with Mask"
    # Force transfer to channel's output buffer
    flush $telnetOut

  }
} else {
  puts $telnetOut "Problem with digital outputs"
  # Force transfer to channel's output buffer
  flush $telnetOut
}
# Close socket
TCP_CloseSocket $socketID
```

This is what gets displayed on a telnet window for the above example.



7.4 Gathering with motion

7.4.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

7.4.2 Description

This example opens a TCP connection, kills the single axis group, then initializes and homes the group. The program then configures the parameters for the gathering (data to be collected: setpoint and current position). It then defines an action (GatheringRun) to an event (SGamma.MotionStart). When the positioner moves from 0 to 50, the data is gathered (with a divisor equal to 100, data is collected every 100th servo cycle, or every 10 ms). At the end, the gathering is stopped and saved in a text file (*Gathering.dat* in Admin/Public directory of the controller). Finally, the program ends by closing the socket.

Please see sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

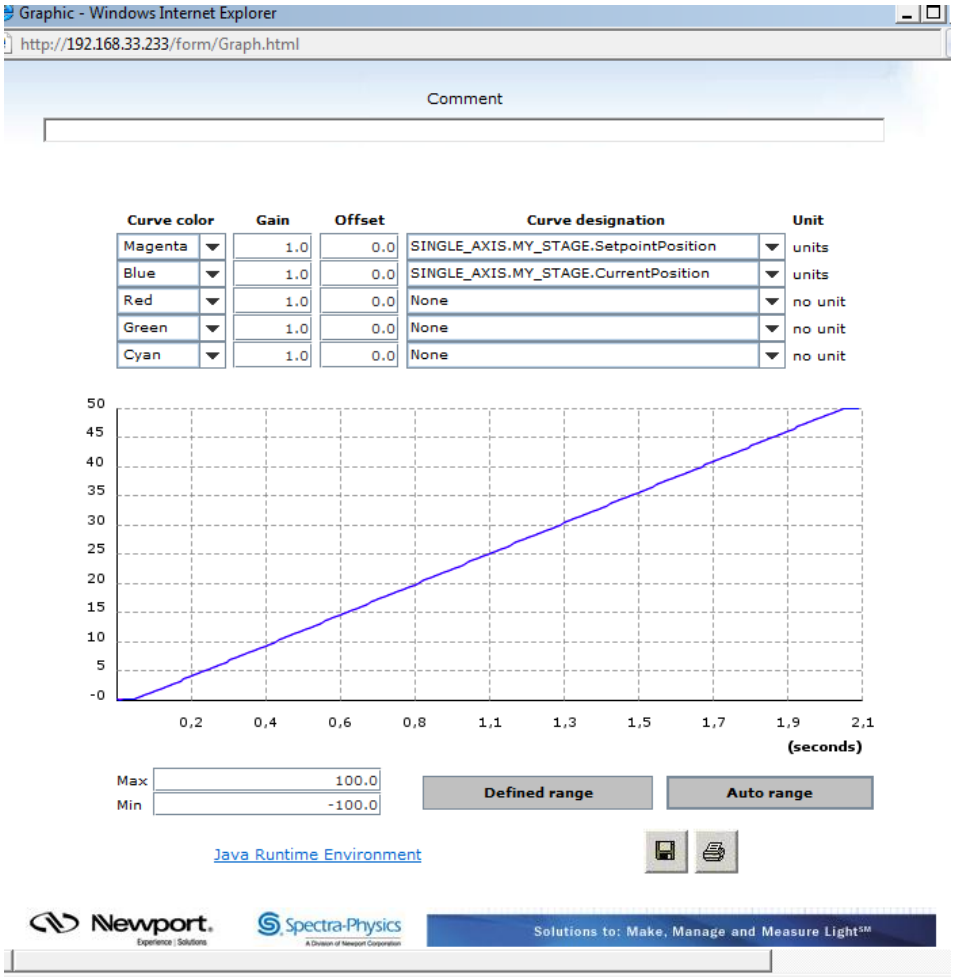
7.4.3 Tcl Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE AXIS.MY STAGE"
set Type1 "SINGLE_AXIS.MY_STAGE.SetpointPosition"
set Type2 "SINGLE_AXIS.MY_STAGE.CurrentPosition"
set Event "SGamma.MotionStart"
set Action "GatheringRun"
```

```
set Displacement 50
set NbPoints 1000
set Div 100
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      # Kill group
      set code [catch "GroupKill $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupKill" $telnetOut
            return
      }

      # Initialize group
      set code [catch "GroupInitialize $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
            return
      }

      # Home group
      set code [catch "GroupHomeSearch $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }

      # Configure gathering parameters
      set code [catch "GatheringConfigurationSet $socketID $Type1
$Type2"]
      if {$code != 0} {
    DisplayErrorAndClose $socketID $code "GatheringConfigurationSet"
$telnetOut
            return
      }

      # Add an event
      set code [catch "EventAdd $socketID $Positioner $Event 0
                $Action $NbPoints $Div   0"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "EventAdd" $telnetOut
            return
      }

      # Move positioner
      set code [catch "GroupMoveRelative $socketID $Group $Displacement"]
      if {$code != 0} {
```

```
            DisplayErrorAndClose $socketID $code "GroupMoveRelative"
$telnetOut
            return
      }

      # Stop gathering and save data
      set code [catch "GatheringStopAndSave $socketID"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GatheringStopAndSave"
$telnetOut
      return
      }

      # Close TCP socket
      set code [catch "TCP_CloseSocket $socketID"]
}
```

When pressing the *Gathering display* button of the terminal window of the XPS web site interface, the following data is displayed:



7.5 External gathering

7.5.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

7.5.2 Description

This example opens a TCP connection, kills the single axis group, then initializes and homes the group. The program then configures the parameters for the external gathering (data to be collected: ExternalLatchPosition and GPIO2.ADC1 value). It defines an action (ExternalGatheringRun) to an event (Immediate). Each time the trigger in receives a signal, the data is gathered (with a divisor equal to 1, gathering takes place every signal on the trigger input). Every second, the current number of gathered data points is displayed. At the end, the external gathering is stopped and saved in a text file (*ExternalGathering.dat* in Admin/Public directory of the controller). Finally, the program ends by closing the socket.

Please see **Section 6**, ***Proposed function for error handling.***

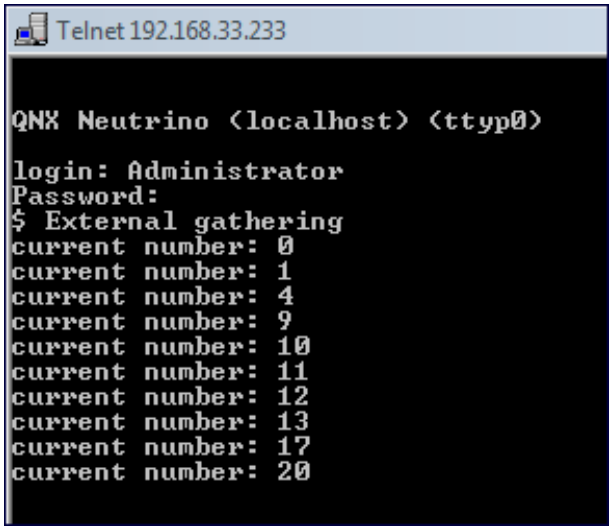
7.5.3 Tcl Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}
# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE AXIS.MY STAGE"
set Type1 "SINGLE_AXIS.MY_STAGE.ExternalLatchPosition"
set Type2 "GPIO2.ADC1"
set Event "Immediate"
set Action "ExternalGatheringRun"
set NbPoints 20
set Div 1
set Current 0
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      # Kill group
      set code [catch "GroupKill $socketID $Group"]
      if {$code != 0} {
```

```
            DisplayErrorAndClose $socketID $code "GroupKill" $telnetOut
            return
      }

      # Initialize group
      set code [catch "GroupInitialize $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
            return
      }

      # Home group
      set code [catch "GroupHomeSearch $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }

      # Configure gathering parameters
      set code [catch "GatheringExternalConfigurationSet $socketID $Type1
$Type2"]
      if {$code != 0} {
DisplayErrorAndClose $socketID $code "GatheringExternalConfigurationSet"
$telnetOut
            return
      }

      # Add an event
set code [catch "EventAdd $socketID $Positioner $Event 0 $Action
$NbPoints $Div   0"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "EventAdd" $telnetOut
            return
      }

      # Push on TRIG IN button…
      puts $telnetOut "External gathering"
      # Force transfer to channel's output buffer
      flush $telnetOut

      # Wait end of external gathering
      while {$Current < $NbPoints} {

            # Get current acquired point number
            set code [catch "GatheringExternalCurrentNumberGet $socketID
Current Max"]
            if {$code != 0} {
                  DisplayErrorAndClose $socketID $code
"GatheringExternalCurrentNumberGet" $telnetOut
                  return
            } else {
                  puts $telnetOut "current number: $Current"
                  # Force transfer to channel's output buffer
                  flush $telnetOut
```

```
                after 1000
            }
        }

        # Stop external gathering and save data
        set code [catch "GatheringExternalStopAndSave $socketID"]
        if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"GatheringExternalStopAndSave" $telnetOut
            return
        }
        # Close TCP socket
        set code [catch "TCP_CloseSocket $socketID"]
}
```

This is what gets displayed in a Telnet window for the above example and when the trigger in receives a signal every second. For details about Telnet connections, see **Section 4** ***Principle of a Tcl script redirection to a telnet session***:



7.6 Position Compare

7.6.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

7.6.2 Description

This example opens a TCP connection, kills the single axis group, then initializes and homes the group. With an absolute move, the positioner moves to the start position –15. Then, the program configures the parameters for the position compare (enabled from -10 to +10 with step position of 1 unit). It enables the position compare functionality and executes a relative move of 25 (final position will be –15+25 = +10). During this move, between the positions –10 and +10, pulses are sent by the trigger output for each 1 unit incremental position. The position compare mode is then disabled and the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.6.3 Tcl Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE_AXIS.MY_STAGE"
set StartPosition -15
set Displacement 25
set MinPos -10
set MaxPos 10
set StepPos 1
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      # Kill group
      set code [catch "GroupKill $socketID $Group"]
      if {$code != 0} {
```

```
            DisplayErrorAndClose $socketID $code "GroupKill"
            return
      }
      # Initialize group
      set code [catch "GroupInitialize $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
            return
      }
      # Home group
      set code [catch "GroupHomeSearch $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }
      # Move positioner to start position
      set code [catch "GroupMoveAbsolute $socketID $Group
$StartPosition"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupMoveAbsolute"
$telnetOut
            return
      }
      # Set position compare parameters
      set code [catch "PositionerPositionCompareSet $socketID $Positioner
$MinPos   $MaxPos $StepPos"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"PositionerPositionCompareSet" $telnetOut
            return
      }

      # Enable position compare mode
      set code [catch "PositionerPositionCompareEnable $socketID
$Positioner"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"PositionerPositionCompareEnable" $telnetOut
            return
      }

      # Move positioner
      set code [catch "GroupMoveRelative $socketID $Group $Displacement"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupMoveRelative"
$telnetOut
            return
      }
      # Disable position compare mode
      set   code   [catch   "PositionerPositionCompareDisable   $socketID
$Positioner"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"PositionerPositionCompareDisable" $telnetOut
            return
```

```
      }
      # Close TCP socket
      set code [catch "TCP_CloseSocket $socketID"]
}
```

7.7 Master-Slave Mode

7.7.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE
XY	1	XY	XY.X and XY.Y

7.7.2 Description

This example opens a TCP connection, kills the Singles Axis group and the XY group. The program then initializes and homes the groups. It sets the parameters for the master slave mode (slave: single axis group, master: X positioner from XY group). Then, it enables the master slave mode and executes a relative move of 20 units with the master positioner. At the same time, the slave positioner executes the same move as the master. The master slave mode is then disabled and the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.7.3 TCL Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}
# Initialization
set TimeOut 60
set SlaveGroup "SINGLE_AXIS"
set XYGroup "XY"
set MasterPositioner "XY.X"
set MasterRatio 1
set code 0
set Displacement 20
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts stdout "OpenConnection failed => $code"
} else {
      # Kill single axis group
      set code [catch "GroupKill $socketID $SlaveGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "Single axis GroupKill"
```

```
            return
      }

      # Initialize single axis group
      set code [catch "GroupInitialize $socketID $SlaveGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "Single axis
GroupInitialize" $telnetOut
            return
      }

      # Home single axis group
      set code [catch "GroupHomeSearch $socketID $SlaveGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "Single axis
GroupHomeSearch" $telnetOut
            return
      }

      # Kill XY group
      set code [catch "GroupKill $socketID $XYGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "XY GroupKill"
$telnetOut
            return
      }

      # Initialize XY group
      set code [catch "GroupInitialize $socketID $XYGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "XY GroupInitialize"
$telnetOut
            return
      }

      # Home XY group
      set code [catch "GroupHomeSearch $socketID $XYGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "XY GroupHomeSearch"
$telnetOut
            return
      }

      # Set slave (single axis group) with its master
      # (positioner from any group : XY here)
      set code [catch "SingleAxisSlaveParametersSet $socketID $SlaveGroup
$MasterPositioner $MasterRatio"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"SingleAxisSlaveParametersSet" $telnetOut
            return
      }

      # Enable master-slave mode (group must be ready)
      set code [catch "SingleAxisSlaveModeEnable $socketID $SlaveGroup"]
      if {$code != 0} {
```

```
            DisplayErrorAndClose $socketID $code
"SingleAxisSlaveModeEnable" $telnetOut
            return
      }

      # Move master positioner
      # (the slave must follow the master in relation to a ratio)
      set code [catch "GroupMoveRelative $socketID $MasterPositioner
$Displacement"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupMoveRelative"
$telnetOut
            return
      }

      # Disable master-slave mode
      set code [catch "SingleAxisSlaveModeDisable $socketID $SlaveGroup"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"SingleAxisSlaveModeDisable" $telnetOut
            return
      }

      # Close TCP socket
      set code [catch "TCP_CloseSocket $socketID"]
}
```

7.8 Jogging

7.8.1 Configuration

Group type	Number	Group name	Positioner name
XY	1	XY	XY.X and XY.Y

7.8.2 Description

This example opens a TCP connection, kills the XY group, then initializes and homes the group. It enables the jog mode and sets the parameters to move a positioner in the positive direction with a velocity of 10 units/s for 1.5 seconds. Then, during the 2 next seconds, the positioner moves in the reverse direction with a velocity of -20 units/s, and finally stops (velocity set to 0). The jog functionality is then disabled and the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.8.3 TCL code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl argv(0)" r+]
} else {
set telnetOut stdout
}
# Initialization
set TimeOut 60
set Group "XY"
set Positioner "XY.X"
set Velocity1 10
set Velocity2 -20
set Acceleration 80
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      # Kill group
      set code [catch "GroupKill $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupKill" $telnetOut
            return
      }

      # Initialize group
```

```
      set code [catch "GroupInitialize $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
            return
      }

      # Home group
      set code [catch "GroupHomeSearch $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }

      # Enable jog mode (group must be ready)
      set code [catch "GroupJogModeEnable $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupJogModeEnable"
$telnetOut
            return
      }

      # Set jog parameters to move a positioner => constant velocity is
not null
      set code [catch "GroupJogParametersSet $socketID $Positioner
$Velocity1 $Acceleration"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
            return
      }

      # Wait 3 seconds
      after 3000

      # Set jog parameters to move the positioner in the reverse sense
      # => constant  velocity is not null
      set code [catch "GroupJogParametersSet $socketID $Positioner
$Velocity2 $Acceleration"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
            return
      }

      # Wait 3 seconds
      after 3000

      # Set jog parameters to stop a positioner => constant velocity is
null
      set code [catch "GroupJogParametersSet $socketID $Positioner 0
$Acceleration"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
```

```
            return
      }

      # Disable jog mode
      # (constant velocity must be null on all positioners from group)
      set code [catch "GroupJogModeDisable $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupJogModeDisable"
$telnetOut
            return
      }

      # Close TCP socket
      set code [catch "TCP_CloseSocket $socketID"]
}
```

7.9 Jogging and Gathering

7.9.1 Configuration

Group type	Number	Group name	Positioner name
XY	1	XY	XY.X_VP and XY.Y_VP

7.9.2 Description

This example opens a TCP connection, kills the XY group, then initializes and homes. The program then configures the parameters for gathering (data to be collected: setpoint position, current position, setpoint velocity and setpoint acceleration). It displays the maximum acquisition per type of data that can be collected (max total data acquisition/number of data types = 1000000/4 = 250000). It defines an action (GatheringRun) to an event (Immediate). When the jog mode is enabled, it changes the jog speed and acceleration. At the end, the jog mode is disabled, the gathering is stopped and saved in a text file (*Gathering.dat* in Admin/Public directory of the controller). Finally, the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.9.3 Tcl Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}
# Initialization
set TimeOut 120
set Moteur "XY"
set Mot "XY.X"
set A "XY.X.SetpointPosition"
set B "XY.X.CurrentPosition"
```

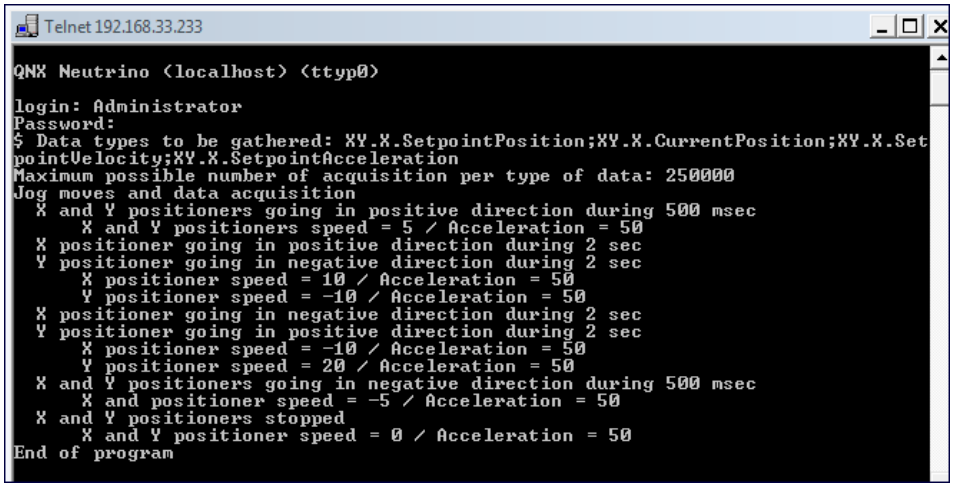
```
set C "XY.X.SetpointVelocity"
set D "XY.X.SetpointAcceleration"
set Event "Immediate"
set Action "GatheringRun"
set Num 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      # Force transfer to channel's output buffer
      flush $telnetOut
} else {
      # Kill group
      set code [catch "GroupKill $socketID $Moteur"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code " $telnetOut
            return
      }

      # Initialize group
      set code [catch "GroupInitialize $socketID $Moteur"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
            return
      }

      # Home group
      set code [catch "GroupHomeSearch $socketID $Moteur"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }

      # Set gathering parameters
      set code [catch "GatheringConfigurationSet $socketID $A $B $C $D"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"GatheringConfigurationSet" $telnetOut
            return
      }

      # Get gathering parameters
      set code [catch "GatheringConfigurationGet $socketID J"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"GatheringConfigurationGet" $telnetOut
            return
      } else {
            puts $telnetOut "Data types to be gathered: $J"
            # Force transfer to channel's output buffer
            flush $telnetOut
      }
```

```
      # Get gathering current acquired point number
      set code [catch "GatheringCurrentNumberGet $socketID Num Max"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"GatheringCurrentNumberGet" $telnetOut
            return
      } else {
            puts $telnetOut "Maximum possible number of acquisition
            per type of data: $Max"
            # Force transfer to channel's output buffer
            flush $telnetOut
      }

      # Add an event
      set code [catch "EventAdd $socketID $Mot $Event 0 $Action 20000 10
0"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "EventAdd" $telnetOut
            return
      }

      # Enable jog mode
      set code [catch "GroupJogModeEnable $socketID $Moteur"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupJogModeEnable"
$telnetOut
            return
      }

      # Wait 2 seconds
      after 2000
      puts $telnetOut "Jog moves and data acquisition"
      # Force transfer to channel's output buffer
      flush $telnetOut

      # Set jog parameters to move both positioners in the positive
      # direction
      set code [catch "GroupJogParametersSet $socketID $Moteur 5 50 5
50"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
      return
      }
      puts $telnetOut "  X and Y positioners going in positive
direction during 500 msec"
      flush $telnetOut
      puts $telnetOut " X and Y positioners speed = 5 / Acceleration =
50"
      flush $telnetOut
      # Wait 500 milliseconds
      after 500
      # Set jog parameters to move both positioners,
      # the first in the positive  direction, the second in the
      # negative
```

```
      set code [catch "GroupJogParametersSet $socketID $Moteur 10 50 -10
50"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
      return
      }
      puts $telnetOut " X positioner going in positive direction during 2
sec"
      flush $telnetOut
      puts $telnetOut " Y positioner going in negative direction during 2
sec"
      flush $telnetOut
      puts $telnetOut " X positioner speed = 10 / Acceleration = 50"
      flush $telnetOut
      puts $telnetOut " Y positioner speed = -10 / Acceleration = 50"
      flush $telnetOut
      # Wait 2 seconds
      after 2000
      # Set jog parameters to move both positioners in the reverse
      # sense
      set code [catch "GroupJogParametersSet $socketID $Moteur -10 50 20
50"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
            return
      }
      puts $telnetOut " X positioner going in negative direction during 2
sec"
      flush $telnetOut
      puts $telnetOut " Y positioner going in positive direction during 2
sec"
      flush $telnetOut
      puts $telnetOut " X positioner speed = -10 / Acceleration = 50"
      flush $telnetOut
      puts $telnetOut " Y positioner speed = 20 / Acceleration = 50"
      flush $telnetOut
      # Wait 2 seconds
      after 2000
      # Set jog parameters to move both positioners in the negative
      # direction
      set code [catch "GroupJogParametersSet $socketID $Moteur -5 50 -5
50"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupJogParametersSet"
$telnetOut
      return
      }
puts $telnetOut "X and Y positioners going in negative direction during
500 msec"
      flush $telnetOut
      puts $telnetOut "X and positioner speed = -5 / Acceleration = 50"
      flush $telnetOut
      # Wait 500 milliseconds
      after 500
```

```
      # Set jog parameters to stop the positioners => constant
      # velocities are null
      set code [catch "GroupJogParametersSet $socketID $Moteur 0 50 0
50"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code
"GroupJogParametersSet"$telnetOut
      return
      }
      puts $telnetOut " X and Y positioners stopped"
      flush $telnetOut
      puts $telnetOut " X and Y positioner speed = 0 / Acceleration = 50"
      flush $telnetOut
      # Wait 500 milliseconds
      after 500
      # Disable jog mode
      set code [catch "GroupJogModeDisable $socketID $Moteur"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GroupJogModeDisable"
$telnetOut
      return
      }
      # Stop gathering and save data
      set code [catch "GatheringStopAndSave $socketID"]
      if {$code != 0} {
      DisplayErrorAndClose $socketID $code "GatheringStopAndSave"
$telnetOut
      return
      }
      # Close socket
      puts $telnetOut "End of program"
      flush $telnetOut
      TCP_CloseSocket $socketID
}
```

This is what gets displayed on a Telnet window for the above example. For details about Telnet connections, see **Section 4**, ***Principle of a Tcl script redirection to a telnet session***.



7.10 Analog Position Tracking

7.10.1 Configuration

Group type	Number	Group name	Positioner name
XY	1	XY	XY.X and XY.Y

7.10.2 Description

This example opens a TCP connection, kills the XY group, then initializes and homes the group. It sets the parameters for the position analog tracking functionality (positioner, analog input, offset, scale, velocity and acceleration) and enables the analog tracking mode. The mode gets activated for 20 seconds. During this time, the stage position follows the voltage of the analog input GPIO2.ADC1. Then, the analog tracking mode gets disabled and the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.10.3 TCL Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 60
set Group "XY"
set Positioner "XY.X"
set AnalogInput "GPIO2.ADC1"
set Offset 0
set Scale 1
set Velocity 20
set Acceleration 80
set TrackingType "Position"
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
puts stdout "OpenConnection failed => $code"
} else {
      # Kill group
      set code [catch "GroupKill $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupKill" $telnetOut
            return
      }
```

```
      # Initialize group
      set code [catch "GroupInitialize $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
            return
      }
      # Home group
      set code [catch "GroupHomeSearch $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }
      # Set analog tracking parameters
      set code [catch "PositionerAnalogTrackingPositionParametersSet
            $socketID   $Positioner $AnalogInput $Offset
            $Scale $Velocity $Acceleration"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "PositionerAnalogTrackingPositionParametersSet" $telnetOut
            return
      }
      # Enable analog position tracking mode (group must be ready)
      set code [catch "GroupAnalogTrackingModeEnable $socketID $Group
            $TrackingType"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "GroupAnalogTrackingModeEnable" $telnetOut
            return
      }

      # Change the amplitude of GPIO2.ADC1 analog input during 20 seconds
      after 20000
      # Disable analog position tracking mode
      set code [catch "GroupAnalogTrackingModeDisable $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
            "GroupAnalogTrackingModeDisable" $telnetOut
            return
      }
      # Close TCP socket
      puts $telnetOut "End of program"
      flush $telnetOut
      set code [catch "TCP_CloseSocket $socketID"]
}
```

7.11 Backlash Compensation

7.11.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

7.11.2 Description

This example opens a TCP connection and kills the single axis group. It enables the backlash compensation capability (for this the controller must be in the not initialized state). The group then is initialized and homed. The value of the backlash compensation is set to 0.1. The positioner executes relative moves with the backlash compensation. Finally, the backlash compensation is disabled and the program ends by closing the socket.

CAUTION

- **The HomeSearchSequenceType in the stages.ini file must not be set as CurrentPositionAsHome.**
- **The Backlash parameter in the stages.ini file must be greater than zero.**
- **To apply any modifications of the stages.ini, the controller must be rebooted after the modification is made.**

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.11.3 TCL Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 60
set Group "SINGLE_AXIS"
set Positioner "SINGLE_AXIS.MY_STAGE"
set BacklashValue 0.1
set Displacement 10
set code 0
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
       puts stdout "OpenConnection failed => $code"
       flush $telnetOut
} else {
```

```
      # Kill group
      set code [catch "GroupKill $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupKill"
            return
      }



# Enable backlach compensation
#####################################################################
# CAUTION :                                                         #
# Group must be "not initialized" and Backlash>0 in the             #
# "stages.ini" file                                                 #
#####################################################################
      set code [catch "PositionerBacklashEnable $socketID $Positioner"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"PositionerBacklashEnable" $telnetOut
            return
      }
      # Initialize group
      set code [catch "GroupInitialize $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
            return
      }
      # Home group
      set code [catch "GroupHomeSearch $socketID $Group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
            return
      }
      # Modify Backlash value
      # Caution : Backlash > 0 in the file "stages.ini"
      set code [catch "PositionerBacklashSet $socketID $Positioner
$BacklashValue"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "PositionerBacklashSet"
$telnetOut
            return
      }
      # Move group in positive direction
      set code [catch "GroupMoveRelative $socketID $Group $Displacement"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupMoveRelative"
$telnetOut
            return
      }
      # Move group in negative direction
      set code [catch "GroupMoveRelative $socketID $Group -
$Displacement"]
      if {$code != 0} {
```

```
            DisplayErrorAndClose $socketID $code "GroupMoveRelative"
$telnetOut
            return
      }
      # Disable Backlash (if you want to do trajectory, jogging or
      # tracking)
      # CAUTION : to enable backlash, you must call "GroupKill" or
      # "KillAll" to come back in "not initialized" status
      set code [catch "PositionerBacklashDisable $socketID $Positioner"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code
"PositionerBacklashDisable" $telnetOut
            return
      }
      # Close TCP socket
      puts $telnetOut "End of program"
      flush $telnetOut
      set code [catch "TCP_CloseSocket $socketID"]
}
```

7.12 Timer Event and Global Variables

7.12.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

7.12.2 Description

The script *StartScript.tcl* opens a TCP connection, configures a timer and uses this timer as an event. The action, in relation to this timer event, executes a second Tcl script named *MyScript.tcl*. The *StartScript.tcl* script sets a global variable and closes the socket.

The timer is a permanent event. The frequency of the timer is set by the divisor, in this example 20000, which means that the second Tcl script gets executed every 20000th servo loop or every 2 seconds (divisor/servo loop rate = 20000/10000 = 2 seconds).

The script *MyScript.tcl* reads the global variable, increments it as long as the variable is below 10. When the global variable is equal to 10, the second script deletes the timer event and finally, the program ends by closing the socket.

Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.12.3 TCL Code

StartScript.tcl

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}
```

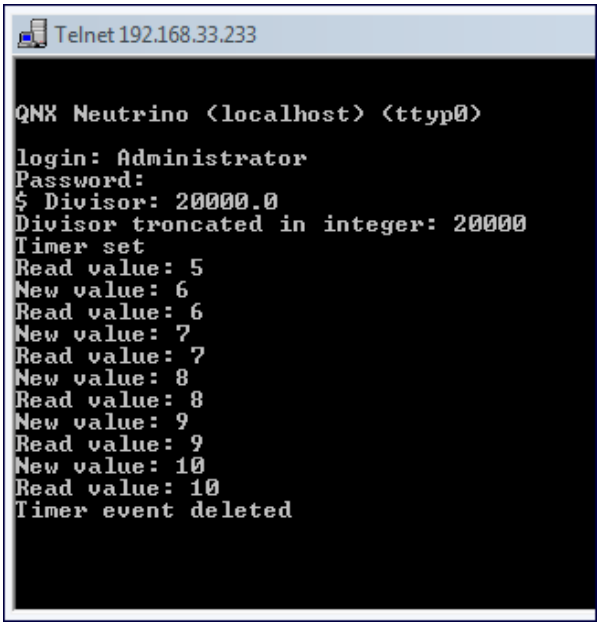
```
# Initialization
set TCPTimeOut 60
set code 0
set ISRPeriodSec 0.0001
set Positioner "SINGLE_AXIS.MY_STAGE"
set TimerName "Timer1"
set TimerPeriodSec 2
set EvtParam 0
set Action "ExecuteTCLScript"
set TCLFile "MyScript.tcl"
set TCLTask "MyTask"
set GlobalVarNumber 1
set Value 5
# Open TCP socket
set code [catch "OpenConnection $TCPTimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      flush $telnetOut
} else {
      # Calculate divisor (periods are in seconds)
      set Divisor [expr {$TimerPeriodSec / $ISRPeriodSec}]
      puts $telnetOut "Divisor: $Divisor"
      flush $telnetOut
      set Divisor [expr {int($Divisor)} ]
      puts $telnetOut "Divisor troncated in integer: $Divisor"
      flush $telnetOut
      # Configure a timer
      set code [catch "TimerSet $socketID $TimerName $Divisor"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "TimerSet" $telnetOut
            return
      } else {
            puts $telnetOut "Timer set"
            flush $telnetOut
      }
      # Add timer event with an action that allows to execute
      # "MyScript.tcl"
      set code [catch "EventAdd $socketID $Positioner $TimerName
      $EvtParam $Action $TCLFile $TCLTask $tcl_argv(0)"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "EventAdd" $telnetOut
            return
      }
      # Set global variable
      set code [catch "GlobalArraySet $socketID $GlobalVarNumber $Value"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GlobalArraySet"
$telnetOut
            return
      }
      # close TCP socket
      set code [catch "TCP CloseSocket $socketID"]
}
```

MyScript.tcl

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl argv(0) != 0} {
set telnetOut [ open "/dev/$tcl_argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TCPTimeOut 60
set code 0
set GlobalVarNumber 1
set ReadValue 0
set NewValue 0
set END 10
set Positioner "SINGLE_AXIS.MY_STAGE"
set EventName "Timer1"
set EventPara 0
# Open TCP socket
set code [catch "OpenConnection $TCPTimeOut socketID"]
if {$code != 0} {
puts  "OpenConnection failed => $code"
} else {
      # Read global variable
      set code [catch "GlobalArrayGet $socketID $GlobalVarNumber ReadValue"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GlobalArrayGet" $telnetOut
            return
      } else {
            puts $telnetOut "Read value: $ReadValue"
            flush $telnetOut
      }

      if {$ReadValue < $END} {
            # Increment global variable
            set NewValue [expr {$ReadValue + 1}]
            # Set global variable to a new value
      set code [catch "GlobalArraySet $socketID $GlobalVarNumb $NewValue"]
            if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GlobalArraySet" $telnetOut
                  return
            } else {
                  puts $telnetOut "New value: $NewValue"
                  flush $telnetOut
            }
      } else {
            # Delete timer event
            set code [catch "EventRemove $socketID $Positioner $EventName
$EventPara"]
            if {$code != 0} {
```

```
                DisplayErrorAndClose $socketID $code "EventRemove" $telnetOut
                return
            } else {
                puts $telnetOut "Timer event deleted"
                flush $telnetOut
            }
        }
        # close TCP socket
        set code [catch "TCP_CloseSocket $socketID"]
}
```

This is what gets displayed on a Telnet window for the above example. For details about Telnet connections, see **Section 4,** ***Principle of a Tcl script redirection to a telnet session*** :



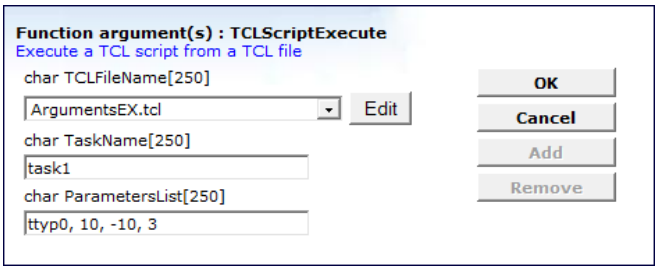
7.13 Tcl Script with Input Arguments

7.13.1 Configuration

Group type	Number	Group name	Positioner name
Single axis	1	SINGLE_AXIS	SINGLE_AXIS.MY_STAGE

7.13.2 Description

This example opens a TCP connection, kills the single axis group, then initializes and homes the group. The program reads the three required input arguments: the start position, the end position, and the number of cycles for moving from the target position to the end position. When the user enters the arguments 10, -10 and 3 via the web site interface, the positioner moves from –10 to +10 three times. Then, the program ends by closing the socket.



Please see the sections:

4 Principle of a Tcl script redirection to a telnet session.

6 Proposed function for error handling.

7.13.3 TCL Code

```
# Set channel's name to be used for telnet.
# In this example we assume it is passed to the script as the
# first argument, if not specified output to stdio.
# Open the channel for read mode and get its id,
# this is the id that will be passed to puts function.
if {$tcl_argv(0) != 0} {
set telnetOut [ open "/dev/$tcl argv(0)" r+]
} else {
set telnetOut stdout
}

# Initialization
set TimeOut 20
set code 0
set group "SINGLE_AXIS"
# Open TCP socket
set code [catch "OpenConnection $TimeOut socketID"]
if {$code != 0} {
      puts $telnetOut "OpenConnection failed => $code"
      flush $telnetOut
} else {
      # Recover the input arguments entered by the user
      # In this case argument 0 is used for Telnet channel identifier
      if {$tcl_argc == 4} {
            set startpos $tcl_argv(1)
            set endpos $tcl argv(2)
            set cycles $tcl_argv(3)
      } else {
            puts $telnetOut "Wrong number of parameters, argument
                  1, 2 and 3 are needed"
            flush $telnetOut
            set code [catch "TCP_CloseSocket $socketID"]
            return
      }

      # Kill group
      set code [catch "GroupKill $socketID $group"]
      if {$code != 0} {
            DisplayErrorAndClose $socketID $code "GroupKill" $telnetOut
            return
```

```
	}
	# Initialize group
	set code [catch "GroupInitialize $socketID $group"]
	if {$code != 0} {
		DisplayErrorAndClose $socketID $code "GroupInitialize"
$telnetOut
		return
	}
	# Home group
	set code [catch "GroupHomeSearch $socketID $group"]
	if {$code != 0} {
		DisplayErrorAndClose $socketID $code "GroupHomeSearch"
$telnetOut
		return
	}
	# Loop until the number of cycles
	for { set i 0} {($i < $cycles) } {incr i} {
		# Move group to start position
		set code [catch "GroupMoveAbsolute $socketID $group
$startpos"]
		if {$code != 0} {
		DisplayErrorAndClose $socketID $code "GroupMoveAbsolute"
$telnetOut
			return
		}
		# Move group to end position
		set code [catch "GroupMoveAbsolute $socketID $group $endpos"]
		if {$code != 0} {
		DisplayErrorAndClose $socketID $code "GroupMoveAbsolute"
$telnetOut
			return
		}
	}
	# Close TCP socket
	set code [catch "TCP_CloseSocket $socketID"]
}
```

Service Form

Your Local Representative

Tel.: _____

Fax: _____

Name: _____

Company: _____

Address: _____

Country: _____

P.O. Number: _____

Item(s) Being Returned: _____

Model#: _____

Return authorization #: _____
(Please obtain prior to return of item)

Date: _____

Phone Number: _____

Fax Number: _____

Serial #: _____

Description: _____

Reasons of return of goods (please list any specific problems): _____

Newport®

Visit Newport Online at:
www.newport.com

North America & Asia
Newport Corporation
1791 Deere Ave.
Irvine, CA 92606, USA

Sales
Tel.: (800) 222-6440
e-mail: sales@newport.com

Technical Support
Tel.: (800) 222-6440
e-mail: tech@newport.com

Service, RMAs & Returns
Tel.: (800) 222-6440
e-mail: service@newport.com

Europe
MICRO-CONTROLE Spectra-Physics S.A.S
9, rue du Bois Sauvage
91055 Évry CEDEX
France

Sales
Tel.: +33 (0)1.60.91.68.68
e-mail: france@newport.com

Technical Support
e-mail: tech_europe@newport.com

Service & Returns
Tel.: +33 (0)2.38.40.51.55

mks

Newport® Ophir® Spectra-Physics®