The fusion of ProMAX and Perl

Henry C. Bland

ABSTRACT

ProMAX and Perl are two large software packages which, when combined, offer a practical and powerful processing tool for seismic data processing.

ProMAX is one of the most common seismic data processing software systems currently available. It gained popular acceptance in the processing industry because of its wide array of processing operations, its ability to work with large datasets, its ease of use, and its interactive processing tools. Although a wealth of processing modules are supplied with ProMAX, cases arise when relatively simple but unusual tasks are made daunting by limitations of pre-existing process modules. Working as an add-in module for ProMAX, the CREWES Perl Process lets one write ProMAX processing modules without leaving the ProMAX environment. It fills the niche of solving simple, unusual processing tasks that do not warrant full-scale module development (program coding, menu coding, compiling, and linking). It is ideally suited for *ad hoc* tasks like correcting geometry values in header words or correcting small data errors. With complete access to trace data, it can also be used to implement data processing algorithms. By writing in Perl, one gains access to an extensive library of subroutines that can be harnessed to provide ProMAX with powerful new features. A number of examples illustrate the operation and features of CREWES Perl Process. We show that the fusion of Perl with ProMAX results in a system that is both practical and versatile.

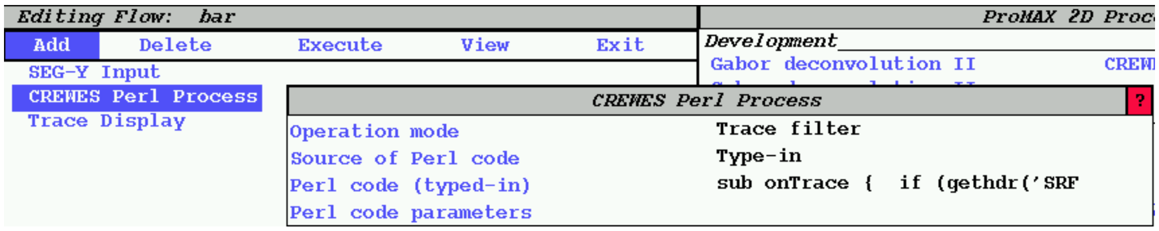
A VERY BRIEF INTRODUCTION TO PERL

Designed to be programmer-friendly and platform-neutral, Perl is a high-level, general-purpose programming language. Perl has become the language of choice across all platforms for programmers engaged in rapid prototyping, system utilities, software tools, system management tasks, database access, graphical programming, and world wide web programming (Christiansen, 1999). It borrows many features of C, but has the ease-of-use of Basic. The standard library of functions is extremely powerful, allowing a few lines of Perl code to do the equivalent of several hundred lines of C code.

In the geophysical processing community, Perl is popular for reformatting and re-working well-log files and acquisition-geometry files. It is an invaluable tool extracting data from text files, and has the ability to read and write binary files. The author is a proponent of Perl, and recommends it to those who work with any kind of data. There are many excellent sources of information about Perl on the web (www.perl.com) and in printed literature (Wall, 2002 and Schwartz, 2001). A quick review of the Perl language may aid in the understanding of the examples that follow.

THE CREWES PERL PROCESS

The CREWES Perl Process (CPP) is an add-on module, which integrates into the ProMAX processing environment. When installed, the CREWES Perl Process appears in the menu along with the standard ProMAX processes (e.g. AGC, NMO). CPP takes its direction from user-supplied Perl code, typed directly into its parameter menu. CPP is able to read and/or write both trace headers and trace data under program control. When a flow is executed, each trace filters through CPP and is acted-upon by the user-supplied Perl code. CPP can perform a myriad of trace/data operations based on the cleverness (or simplicity) of the code supplied.



```
Operation mode            Trace filter
Source of Perl code       Type-in
Perl code (typed-in)      sub onTrace {  if (gethdr('SRF
Perl code parameters
```

FIG. 1. CPP can be inserted into any processing flow. The figure shows some of the parameters available.

Perl code can come from one of two places. It can be typed directly into the CPP parameter menu, or it can be read from a file (external to ProMAX). In cases where the Perl code is lengthy, it is often more convenient to work with an external file due to improved editing facilities.

MODIFYING TRACE HEADERS

ProMAX trace headers are typically manipulated using *Trace Header Math* process. CPP can do much more than simply modify trace headers, but to learn the basics of CPP it is helpful to illustrate it as a *Trace Header Math* (THM) replacement. Here are some specific differences between CPP and THM:

- CPP can propagate values from one trace to the next. THM has no memory to allow access to header values from prior traces.
- CPP defines new headers explicitly. THM defines new headers anytime an assignment is made to a previously nonexistent trace header. Though it takes a little more typing to define a new trace header, this prevents accidental creation of new headers in the event of a header-name spelling error.
- CPP automatically converts integer expressions to real values when assigning real header words. It also automatically converts real values to integers when assigning integer header words. In general, one doesn't have to worry about data types with CPP.

Example 1: Simple modification of a trace header

Consider a processing flow like this:

```
Disk Data Input     <- Input
Trace header math  REC_X = REC_X - 8.6
Disk Data Output   -> Output
```

We can accomplish the similar result using CREWES Perl Process in place of the Trace Header Math processes. The new ProMAX flow is:

```
Disk Data Input     <- Input
CREWES Perl Process    (typed-in Perl code to follow)
Disk Data Input     -> Output
```

The following (simplistic) Perl code implements the desired operation (subtract 8.6 from the receiver X-coordinate):

```
sub onTrace {
   $rec_x = gethdr('REC_X');
   $rec_x = $rec_x - 8.5;
   puthdr('REC_X', $rec_x);
}
```

We shall describe the above four lines of Perl code in detail. The first line contains the code: `sub onTrace`. This introduces a subroutine that will do all the work. CPP is very specific about the name of the function that does the work. In *trace mode,* the work function must be called `onTrace`. In *ensemble mode,* the work function must be called `onEnsemble`.

In Example 1, `onTrace` is called once for each trace that is processed. When the `gethdr` function is called (line 2), it reads the `REC_X` header from the current trace. The result is placed in the variable `$rec_x`. In Perl, a dollar sign precedes all variable names[1]. Line 3 subtracts 8.5 from `$rec_x`. Line 4 calls the `puthdr` function to store the `$rec_x` variable into the `REC_X` trace header. Readers might notice that braces (curly brackets) enclose the contents of the `onTrace` subroutine and a semicolon ends every statement. Though we typically indent Perl code with spaces to tabs to make it readable, Perl is insensitive to the presence of extra spaces or tabs. Perl requires semicolons to separate statements, even if they appear on separate lines.

[1] To be more accurate, $ precedes scalar variables and references. Vector variables and "glob" variables are preceded by @ and * respectively.

The Perl motto is “There's more than one way to do it”. We could rewrite Example 1 in a more concise way using the following code:

```
sub onTrace {
   puthdr('REC_X', gethdr('REC_X') - 8.5);
}
```

Example 2: Conditional modification of trace headers

When processing data we often face the challenge of correcting errors in specific header words over a small portion of the dataset. In this example, we conditionally modify the `CDP_Y` header if the `CDP` header is less than four. Consider the following conventional ProMAX processing flow that performs this task:

```
Disk Data Input     <- Input
IF                     Mode: Include the traces in the list
                       Primary Trace: CDP
                       Trace list: 1-4
Trace Header Math      CDP_Y = CDP_Y + 12
ELSE
Trace Header Math      CDP_Y = CDP_Y - 5
END IF
Disk Data Output    -> Output
```

We can accomplish the same task using CREWES Perl Process. The processing flow is trivial:

```
Disk Data Input     <- Input
CREWES Perl Process    (code to follow)
Disk Data Output    -> Output
```

The associated Perl code is shown below:

```
sub onTrace {
      $cdp = gethdr('CDP');
      $cdp_y = gethdr('CDP_Y');
      if ($cdp <= 4) {
         $cdp_y = $cdp_y + 12;
      } else {
         $cdp_y = $cdp_y - 5;
      }
      puthdr('CDP_Y', $cdp_y);
}
```

Again, we could write the previous code snippet more concisely:

```
sub onTrace {
   if (gethdr('CDP') <= 4) {
      puthdr('CDP_Y', gethdr('CDP_Y') + 12);
   } else {
      puthdr('CDP_X', gethdr('CDP_Y') - 5);
   }
}
```

Example 3: Remembering values from trace to trace

One of the weaknesses of ProMAX's *Trace Header Math* is its inability to remember context from one trace to the next. For example, the following *Trace Header Math* operation does not work as expected:

```
CDP = CDP + 1
```

If the `CDP` header is zero for all traces in the dataset, this invocation of *Trace Header Math* would simply cause all traces to have a `CDP` of one. We would not get the more desirable result of a sequence of `CDP` values, increasing by one per trace. Unlike *Trace Header Math*, CPP remembers the value of variables from one trace to the next:

```
$cdp = 0;

sub onTrace {
   $cdp = $cdp + 1;
   puthdr('CDP', $cdp);
}
```

The above CPP code would place an increasing sequence of values in the `CDP` for each trace in the dataset.

Example 4: Applying the Fibonacci series to trace headers

A more interesting demonstration is to assign the Fibonacci series to a trace header for all traces in a dataset. The Fibonacci series is recursively defined in equation (1):

$$fibonacci(x) = \left\{ \begin{array}{lcl} (x=0) \vee (x=1) & : & x \\ otherwise & : & fibonacci(x-1) + fibonacci(x-2) \end{array} \right\} \tag{1}$$

The following example assigns the sequence 1, 2, 3, 5, 8, 13 ... to each trace's SOU_SLOC header word. This task would be is difficult to reproduce using *Trace Header Math*:

```
$f0 = 0;
$f1 = 1;
sub onTrace {
   $sum = $f0 + f1;
   $f0 = $f1;
   $f1 = $sum;
   puthdr('SOU_SLOC',$sum);
}
```

As unrealistic as this example is, it illustrates CPP's ability to perform programming tasks within the ProMAX environment.

Example 5: Defining new header words

CPP requires explicit instructions to create new trace header word. To create a new header word one calls the defhdr function. The defhdr function accepts three arguments: the name of the new header word, the data format (integer, real, or character), and the description. All initialization code, particularly calls to defhdr, must reside outside any subroutines. Details on the defhdr function are presented in the Appendix A.

This following example defines a new trace header that contains the distance from a geophone to the recorder:

```
defhdr('RECODIST','R','Distance to recorder');
$recorderX = 58123;
$recorderY = 872741;

sub onTrace {
   $dx = gethdr('REC_X') - $recorderX;
   $dy = gethdr('REC_Y') - $recorderY;
   puthdr('RECODIST', sqrt(($dx * $dx) + ($dy * $dy)));
}
```

The first three lines are executed during the initialization phase of the flow because they reside outside the onTrace function. It is mandatory to place the defhdr function outside of onTrace, and it makes sense to place the initialization code somewhere where it will only be executed once.

Example 6: Writing character format trace headers

Character headers can be used to display test parameters as generated by the ProMAX *Parameter Test* module. CPP lets us place any kind of textual information into text header words. The following example shows how traces can be annotated based on the value of a particular numerical header word:

```
sub onTrace {
   if ($GEO_COMP == 1) {
      $PARMTEST = 'Vertical';
   } elsif ($GEO_COMP == 2) {
      $PARMTEST = 'Inline';
   } elsif ($GEO_COMP == 3) {
      $PARMTEST = 'Crossline';
   }
}
```

After running CPP with the above code, the `PARMTEST` header would be set to indicate the type of geophone used for each trace. When the output dataset is then viewed using ProMAX's *Trace Display* process, we see value of `PARMTEST` alongside the trace data.

Example 7: Reading character format trace headers

CPP can read from character header words as well as write to them. The ProMAX process called *Floppy Input* reads data from SEG-2 format files[2]. It places the filename in the trace header called `CFILE` and numbers field-file-identifiers (FFIDs) sequentially, regardless of any file number implicit in the filename. The following Perl code can be used to extract the numeric portion of `CFILE` and place it in the FFID. This example assumes that filenames have the form "ABC0000.DAT" where "0000" is the desired file number:

```
sub onTrace {
   $cfile = gethdr('CFILE');
   $ffid = substr($cfile,3,4);
   puthdr('FFID',$ffid);
}
```

We use Perl's `substr` function to extract the portion of the filename that contains the numbers. The code, `substr($cfile,3,4)` means "from `$cfile`, extract 4 characters starting with the character at position 3" (Perl considers the first character position to be zero). The following table illustrates the result of this function.

File name	FFID
ABC1001.DAT	1001
ABC1003.DAT	1003
ZZZ1311.DAT	1311

[2] It is the author's belief that *Floppy Input* is a misnomer, and the process should be called *SEG-2 Input*.

Example 8: Exporting trace headers to a spreadsheet

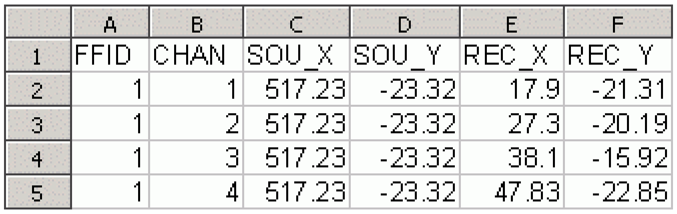
It is often helpful to export trace header information into a spreadsheet package for detailed analysis or graphing. Almost all spreadsheet programs read comma-separated-value format files (CSV files). The following example shows how to create a CSV file from trace headers:

```
open(OUT,”>geom.csv”) || die ”Can’t write geom.csv $!”;
print OUT ”FFID,CHAN,SOU_X,SOU_Y,REC_X,REC_Y\n”;
sub onTrace {
   $ffid = gethdr(’FFID’);
   $chan = gethdr(’CHAN’);
   $sou_x = gethdr(’SOU_Y’);
   $sou_y = gethdr(’SOU_X’);
   $rec_x = gethdr(’REC_X’);
   $rec_y = gethdr(’REC_Y’);
   print OUT ”$ffid,$chan,$sou_x,$sou_y,$rec_x,$rec_y\n”;
}
```

The above example opens the output file *geom.csv* outside the `onTrace` function so that it is opened only once. The last line of the `onTrace` function prints all the header values with a comma separating them. The final `\n` adds a carriage-return to the end of each line. After running CPP, the output file would resemble the following listing:

```
FFID,CHAN,SOU_X,SOU_Y,REC_X,REC_Y
1,1,517.23,-23.32,17.9,-21.31
1,2,517.23,-23.32,27.3,-20.19
1,3,517.23,-23.32,38.1,-15.92
1,4,517.23,-23.32,47.83,-22.85
…etc…
```

If the above file was opened in a spreadsheet program the header values are easy to see:



	A	B	C	D	E	F
1	FFID	CHAN	SOU_X	SOU_Y	REC_X	REC_Y
2	1	1	517.23	-23.32	17.9	-21.31
3	1	2	517.23	-23.32	27.3	-20.19
4	1	3	517.23	-23.32	38.1	-15.92
5	1	4	517.23	-23.32	47.83	-22.85

WORKING WITH ENSEMBLES

Up to this point, all trace operations have been performed a trace at a time. Although CPP allows access the header values of previous trace, it can be useful to look ahead at the next trace in an ensemble.

CPP can operate in *ensemble mode*, where an ensemble of traces become accessible during each call to the user’s work subroutine. When operating in ensemble mode, the work subroutine must be named `onEnsemble`. In addition, the `gethdr` and `puthdr` functions access trace headers as *arrays* instead of scalars.

Example 9: Assigning trace headers in array mode

To illustrate the most basic array operations we will perform some receiver location renumbering. The following example assumes that ensembles of five traces are processed. It will assign the receiver surface location to each of the five traces in each ensemble:

```
sub onEnsemble {
   @stations = (101, 102, 103, 104, 105);
   puthdr('REC_SLOC', @stations);
}
```

This example uses Perl arrays. Perl array variables begin with an ampersand. The array is loaded with values by assignment to a parenthesized list of numbers.

Example 10: A simple array operation with an ensemble of trace headers

The following example reverses all the channels in a shot gather. If the channels were originally sequenced 1,2,3,4,5, the code in the example would reassigned the channels in the sequence 5,4,3,2,1:

```
sub onEnsemble {
   @chans = gethdr('CHAN');
   @reverseChans = reverse(@chans);
   puthdr('CHAN', @reverseChans);
}
```

The `reverse` function is a Perl built-in operator. It accepts an array of values as input, and returns the same array in reverse order.

Example 11: Adjusting an ensemble of trace headers

A more realistic problem to solve is one in which we first scan all values of a particular trace header in an ensemble. Then, based on the scan, we apply a value to all trace headers within the ensemble. The following example, designed for shot records, computes the distance from each trace to the trace nearest the shot. There is a place in the `ROFFSET` header word for each trace:

```
defhdr('ROFFSET','R','Offset to receiver nearest the shot');
sub onEnsemble {
   @offset = gethdr('OFFSET');
   $minOffset = 9999999;
   foreach $o (@offset) {
      if ($o < $minOffset) {
         $minOffset = $o;
      }
   }
   for $i (0..$NTRACES-1) {
      $roffset[$i] = $offset[$i] - $minOffset;
   }
   puthdr('ROFFSET',@roffset);
}
```

The example above introduces two forms of the Perl `for` loop. The first `for` loop sets the variable `$o` to each of the values in the `@offset` array. The second `for` sets the variable `$i` to the sequence of numbers from 0 and `$NTRACES-1` (inclusive). We also introduce the method of accessing an element of an array. When referring to *all* elements of the offset array, we use `@offset`. To access a *single* element of the array we write `$offset[5]` (in this case, giving us the fifth element). Some readers may notice the `$NTRACES` variable seems to come from nowhere. In fact, this variable is one of four *magic variables* that are automatically available to Perl when running within CPP. The following section provides more detail.

MAGIC VARIABLES

The CPP automatically defines some variables for use by Perl code:

`$NUMSMP`	- number of samples per trace
`$SAMPRAT`	- sample interval in units of milliseconds
`$MAXDTR`	- the maximum number of traces per ensemble
`$NTRACES`	- the number of input traces in the current ensemble

The $NUMSMP, $SAMPRAT, and $MAXDTR variables are all preset during the initialization phase (outside onTrace or onEnsemble). The user's initialization Perl code may modify these values, thus altering the dataset parameters for all subsequent processes in the ProMAX flow (for experts only). The $MAXDTR and $NTRACES variables are only meaningful when CPP is operating in ensemble mode. In cases where each ensemble has a different number of traces (such as when processing CDP gathers) the $NTRACES variable will be set to the number of traces in the current ensemble. Altering $NTRACES within an onEnsemble function alters the number of traces output by CPP for the current ensemble. For example, if one were writing a stack function with CPP, the onEnsemble subroutine would set $NTRACES to 1, thus reducing an ensemble of unstacked traces to a single stacked trace.

WORKING WITH SINGLE TRACES OF DATA

All examples to this point have dealt with trace header manipulation. CPP is also able to access and modify trace data. This allows one to write complete data processing modules in Perl using CPP.

Example 12: Reading sample data a trace at a time

When operating in trace mode, 1-D trace data are accessed via a standard Perl array:

```
*trace = traceData;
sub onTrace {
   for $i (0..$NUMSMP-1) {
      $trace[$i] = $trace[$i] * 5 - 2.5;
   }
}
```

The first line assigns the `@trace` array as the conduit for trace data. The * character that appears on line 1 is called a *glob*. It only needs to be used this one time, when calling the `traceData` function. After this assignment, the first sample is accessed using `$trace[0]`, the second as `$trace[1]`, etc. Changes to sample values are automatically propagated to the remainder of the ProMAX flow. To iterate through all the samples, the `for` loop makes use of the `$NUMSMP` magic variable. It contains the number of samples per trace.

The `traceData` function (line 1 of this example) does not look like a function, but it is one. In Perl, functions that require no parameters look like bare words. It is a common mistake to use the C syntax (empty parentheses after the function name) when calling functions without arguments.

Example 13: Simple 1D signal processing

There are situations when overwriting the input trace can cause problems. The following example implements a simple smoothing filter:

```
*inTrace = traceData('<');
*outTrace = traceData('>');
sub onTrace {
   for $i (1 .. $NUMSMP-2) {
      $outTrace[$i] = $inTrace[$i-1] +
      $inTrace[$i] +
      $inTrace[$i+1];
   }
   # special handling of the first and last samples
   $outTrace[0] = $inTrace[0];
   $outTrace[$NUMSMP-1] = $inTrace[$NUMSMP-1];
}
```

This 3-element smoothing filter would not work with a single trace data array because the input trace must remain intact until the smoothing operation is complete. By passing the `‘<’` or `‘>’` argument to the `traceData` function, we define arrays which are designated as input-only and output-only respectively.

WORKING WITH ENSEMBLES OF TRACE DATA

When operating in ensemble mode, CPP can access trace data for the whole ensemble via a two-dimensional array[3]. The same `traceData` function is used to define the array that serves as a data conduit. The left-most array index determines the trace. The right-most array index determines the sample.

The following example reads and writes data to a two-dimensional array of trace data. This particular example computes the DC offset for an ensemble of traces, and subtracts that offset from all trace values.

[3] Two-dimensional arrays are actually arrays of array-references.

```
*trace = traceData;

sub onEnsemble {
   $sum = 0;
   for $t (0..$NTRACES-1) {
      for $i (0..$NUMSMP-1) {
         $sum = $sum + $trace[$t][$s];
      }
   }
   $dcShift = $sum / ($NTRACES * $NUMSMP);
   for $t (0..$NTRACES-1) {
      for $i (0..$NUMSMP-1) {
         $trace[$t][$s] = $trace[$t][$s] - $dcShift;
      }
   }
}
```

FUTURE WORK

CPP is a new module for ProMAX. The latest version reads and writes trace headers and trace data. It currently operates as a filter, operating in trace mode or ensemble mode. We hope to enhance CPP by adding an *input tool* mode of operation. In this mode, CPP can be a trace generator, allowing it to import data directly into ProMAX[4]. Another planned enhancement is to allow access to the ProMAX database and parameter tables.

To benefit fully from the features of CPP, the user requires some knowledge of Perl. Though nearly 1 million programmers already know Perl (O'Reilly, 2002), few geoscientists are currently familiar with it. Enhancements to the CPP documentation, in conjunction with the evolution of a cookbook should help encourage usage of CPP.

CONCLUSION

Through the presentation of several examples, we have introduced some of the features and capabilities of CPP. CPP is still evolving, but it has already proven useful to the researchers within CREWES. CPP's rapid-development capabilities have made it popular for handling a wide variety of ProMAX tasks. We believe this program has a promising future as a practical and versatile problem-solving tool.

REFERENCES

Christiansen, T. and Nathan, T., 1999, Perl FAQ, http://www.perldoc.com/perl5.6/pod/perlfaq.html
Schwartz, L.R. and Phoenix, T., 2001, Learning Perl: O'Reilly and Associates
Wall, L. and Schwartz, R.L., 1991, Programming Perl: O'Reilly and Associates

[4] There is currently a work-around for this missing feature: generate synthetic traces, then replace them with data from an external source

APPENDIX A - FUNCTION REFERENCE

defhdr

Synopsis

```
defhdr($hdrname,$format,$description);
```

Defines new trace headers within the flow. If the trace header already exists, the existing definition is used. If a header definition conflicts with an existing definition due to a mismatched data type, and error occurs.

Parameters

`$hdrname` is the header's name. Trace header names should be no longer than 8 characters. Capitalization is ignored.

`$format` defines the format of the trace header. Possible values are:

`'R'`	Real value (single precision)
`'D'`	Real value (double precision*)
`'I'`	Integer value
`'C'`	Character value

The format of character headers may optionally include a length (the default is 4 characters). For example, define a character trace header of length 12 characters specify a format of 'C12'. Character lengths are always rounded-up to multiples of 4. Avoid defining long character headers, as they may exhaust the overall supply of trace header memory.

*Double precision headers are currently not implemented in most ProMAX modules. Their use is not recommended for compatibility reasons.

`$description` is the header's description. The description can be up to 32 characters in length.

gethdr

Synopsis

```
$value = gethdr($hdrname);
```

```
@values = gethdr($hdrname);
```

In a trace-a-time mode, `gethdr` gets a trace header value from the current trace.

In ensemble mode, `gethdr` gets an array of trace headers for all traces in the current ensemble.

Returns `undef` if the specified header name does not exist.

Parameters

`$hdrname` is the name of the requested trace header.

puthdr

Synopsis

```
puthdr($hdrname, $value);

puthdr($hdrname, @values);
```

Puts a new value in a trace header. In trace mode, a single value is stored in the named header for the current trace. In ensemble mode, the array of values is stored in the current ensemble's traces.

If `$hdrname` does not exist, an error occurs.

Parameters

`$hdrname` is the name of the header to be updated.

`$value` is the value to place in the current trace header (trace mode).

`@value` is an array of header values to place in the current ensemble's trace headers (ensemble mode).

traceData

Synopsis

```
*array1d = traceData($direction)

*array2d = traceData($direction)
```

Designates an array as a conduit for trace data. The returned value is a Perl "glob". In trace mode, the glob allows access to a one-dimensional array of trace data indexed by sample. In ensemble mode, the glob allows access to a two-dimensional array of trace data indexed by trace and sample (in that order).

Parameters

`$direction` (Optional).

If direction is '<' the returned value can be used to access input data. Any alterations to the associated array will be ignored.

If the direction is '>' the returned value can be used to access output data. The data stored within the associated array will be used as the source of output data to the remainder of the ProMAX flow.

If no direction is specified, the returned value can be used to access input data, and it will also be used as the source of output data to the remainder of the ProMAX flow.

If insufficient output data is available (if the array doesn't match the input array's dimensions) the output data is padded with zeros.

The number of traces output (when in ensemble mode) is dependent on the `$NTRACES` magic variable. To reduce the number of traces output, decrease `$NTRACES`. `$NTRACES` should never be increased beyond the value of `$MAXDTR`. The value of MAXDTR is determined by the maximum ensemble size (specified in the ProMAX flow's input tool).

Magic Variables

`$NUMSMP`	- number of samples per trace
`$SAMPRAT`	- sample interval in units of milliseconds
`$MAXDTR`	- the maximum number of traces per ensemble
`$NTRACES`	- the number of input traces in the current ensemble

APPENDIX B – ADDITIONAL EXAMPLES

Example B1: Modify a number of trace headers at the same time

In Perl, there’s more than one way to do it. Here is one novel approach:

```
sub onTrace {
   %coordShift = ( 'REC_X' => 82817,
                   'REC_Y' => 6552512,
                   'SOU_X' => 82817,
                   'REC_Y' => 6552512,
                   'CDP_X' => 82817,
                   'CDP_Y' => 6552512);
   foreach $hdr (keys %coordShift) {
      puthdr($hdr, gethdr($hdr) - $coordShift{$hdr);
   }
}
```

Example B2: Import data from a text file into a trace header

Here we open a file that contains three (white-space separated) columns: CDP, CDP_X and CDP_Y. Next, we read each line of the file, and built two associative arrays of CDP_X coordinate indexed by CDP numbers. Perl will automatically work out the array sizes for us. Finally within our `onTrace` function, we lookup and assign the CDP_X and CDP_Y based on the trace's CDP:

```
open(FILE, "<cdpcoords.txt") || die "Error $!";
while (<FILE>) {
   ($cdp, $cdp_x, $cdp_y) = split(' ');
   $cdp = int($cdp);
   $cdpxByCdp{$cdp} = $cdp_x;
   $cdpyByCdp{$cdp} = $cdp_y;
}
close(FILE);

sub onTrace {
   puthdr('CDP_X',$cdpxByCdp{$CDP});
   puthdr('CDP_Y',$cdpyByCdp{$CDP});
}
```