ETAS

Custom Calculated Signals

User's Guide

DRIVING | EMBEDDED EXCELLENCE

Copyright

Document 010200 V7.2 R02 EN – 06.2018

Contents

1 Introduction

Data analysis may be enhanced by the usage of mathematical calculation rules. To provide the user with a flexible way to define calculation rules an interpreter is used thus allowing users to define their own set of calculation rules upon measured data. Currently, INCA provides Perl — a general-purpose programming language — as an embedded interpreter.

Besides the fact, that Perl has a well-defined and well-documented interface for embedding the interpreter into another application, it is also reasonably fast, and has a rich set of built-in commands producing very compact code.

Perl is expandable with modules. For almost any task a specific module can be found at the *Comprehensive Perl Archive Network* (www.cpan.org).

To compactly store and speedily manipulate large N-dimensional data arrays, there exists a module called *Perl Data Language* (pdl.perl.org). PDL turns Perl into an array-oriented, numerical language similar to such commercial packages as IDL and MatLab.

Note

This manual is intended for experienced Perl programmers only. If you have little or no programming experience in Perl, please refer to learn.perl.org for a brief introduction and for additional references. You might want to read about Perl references (`perlref`) and objects (`perlboot`) as well as the basics of PDL (`PDL::Impatient`).

1.1 Calculated Signals and Perl

Any measured or calculated signal subjected to a calculation rule is represented internally in Perl as a PDL, a 1-dimensional array of doubles. User-defined calculations based on measured data can be implemented by entering a calculation-rule in the dialog 'Define Signal' using simple variables for the signals. In order to distinguish PDLs from other variables and in order to use signal names directly as variable names a special syntax is used: Signal names have to be enclosed by braces and single quotes as, for example, `${´Test_Ch1\VADI-Testdevice:1´}`. A resulting expression may be stored and loaded as part of a configuration in an XML file (file name extension `xcs`).

The usage of such configurations has one essential drawback: because of the fixed operators, the re-use of a configuration is limited. In order to overcome this limitation, custom-operations have been introduced. A custom-operation can be thought of as parameterizable configurations. The calculation-rule, its input parameters and types and some basic documentation is stored together in a Perl module.

2 Custom Operations

2.1 Custom Operation Installation Directories

Custom operations are read from two directories:

- *<EtasShared>*\CalculatedSignals\PM: custom operations provided by ETAS.
- *<EtasData>*\<Product>\CalculatedSignals\PM: custom operations written by the user.

The directories depend on the product (e.g. MDA7.2 or INCA7.2) and the options chosen during installation (paths for EtasData and EtasShared, using shared or isolated installation).

Typical examples would be c:\Etas\EtasShared12\CalculatedSignals\PM or d:\EtasData\MDA7.2\CalculatedSignals\PM.

Within the PM directory the files are organized as follows:

- PM: custom operations relevant to all products.
- PM\online: custom operations relevant to INCA only.
- PM\offline: custom operations relevant to MDA only.
- PM\Templates: sample custom operations that may be copied and modified by the user.
- PM\Calibrations: calibration data used to auto-generate custom operations.

2.2 Basic Structure of a Custom Operation

A Custom Operation is a Perl module, which contains two sections: a code section and a comment section. The delimiters for the comment section are the keywords =pod and =cut.

```
# perl code

=pod
<CUSTOMLIST><!-- XML fragment --> </CUSTOMLIST>
=cut

1;
```

The Perl interpreter only looks at the code section and ignores the comment-section. The code section contains the implementation of the custom operation.

The Calculated Signals dialog only looks at the XML fragment in the comment section and ignores the code section. The XML fragment contains information about which custom operations are available and for each operation a help text and which parameters need to be provided by the user.

2.3 Simple Custom Operation (Without Internal State)

We use the module BinaryAND as a simple example. This module computes for every measure point the bitwise logical AND with a given mask.

```
package BinaryAND;
```

The module starts with the package name in the first line. The package name must be identical to the file-name without the module-extension ".pm".

```
use base qw(CustomCalculationBase);
```

The use base indicates that we are defining a derived Perl object which in this case is derived from the CustomCalculationBase base class which helps implement the custom operation.

```
# $Id: BinaryAND.pm 144577 2009-08-03 14:56:27Z mibosch $
```

The Id-line is a comment, which identifies the module-version.

```
use PDL;
use PDL::Types;
use PDL::Func;
use PDL::Math;
use warnings;
use strict;
```

With the `use` statements other modules are included. Most of them are PDL-specific. The `warnings` module enables all warnings. The `strict` module restricts unsafe constructs. It forces the declaration of every variable with the keyword `my`. The `my` keyword restricts the visibility of a variable to the current module which avoids conflicts between different custom operations.

Any other standard module may be used, as well as any self-defined module, as long, as the modules are stored in the standard Perl-path or in one of the two paths mentioned above.

Below are the two standard methods, which have to be provided by every module.

```
sub new
{
    return (bless {
        _signals => ['input'],
        _parameters => ['mask']
        })->create(@_);
}
```

The purpose of the `new()` method is to create a new instance of a Perl object representing the custom calculation. In Perl an object is a hash table that has been blessed. In this example the curly braces create a reference to a new hash table which has already been initialized with some members. The call to bless marks the hash table as an object of a type determined by the package name (i.e. BinaryAND). Finally the method `create()` is called on our new object. The `create()` method is implemented by the `CustomCalculationBase` base class and finishes the initialization of the object. To do this it needs access to the parameters of the `new()` method, which are passed using `@_` as the argument.

There is usually no need to customize the `new()` method except for the initialization of the `_signals` and `_parameters` members. The `_signals` member is the list of names of the input signals of the custom calculation, and the `_parameters` member is the list of names of the parameters of the custom calculation.

```
sub calc
{
    my ($this, $dim, $input, $mask) = @_;
    return double(long($input) & long($mask));
}
```

The `calc()` method is called internally for evaluation of the custom calculation. The arguments are the object itself (`$this`), as well as the current dimension, i.e. the number of samples to process (`$dim`), then one PDL for each input signal (e.g. `$input`) and finally the parameters (e.g. `$mask`). Using the input signals and parameters the `calc()` method computes a result, which must be a pdl of the same dimension as the input signals. The result is returned as the return value of the `calc()` method.

Note

Because of performance reasons, explicit `for` loops using the `index()` method should be avoided. Instead PDL methods like `slice()` should be used. Many other useful methods for manipulating PDL arrays can be found in the `PDL::Ufunc` module.

The block between `=pod` and `=cut` defines a documentation area, where the XML-description of the custom operation has to be provided. This block may occur only once.

```
=pod

<CUSTOMLIST>

  <VERSION>2.0</VERSION>

  <CUSTOM>
    <FUNCTIONNAME>BinaryAND</FUNCTIONNAME>
    <UNIT>N/A</UNIT>
    <DATATYPE>uint32</DATATYPE>
    <CALCULATIONRULE>BinaryAND::new(${input}, ${mask});</CALCULATIONRULE>
    <DESCRIPTION><![CDATA[bitwise logical AND of <input> with <mask>
parameters:
  <input> = input signal
  <mask>  = bitmask, decimal or hex (0x)]]>
    </DESCRIPTION>
    <FLAGS>NoHistory</FLAGS>
    <PARAMETERLIST>
      <PARAMETER><NAME>${input}</NAME><TYPE>VAR</TYPE></PARAMETER>
      <PARAMETER><NAME>${mask}</NAME><TYPE>INT</TYPE></PARAMETER>
    </PARAMETERLIST>
  </CUSTOM>

</CUSTOMLIST>

=cut
```

The XML description of a Custom Operation is a Custom List, which contains a version and the Custom Operation itself. The version is fixed and should not be changed. The Calculation Rule calls the `new()` method of the object. The order of the signals and parameters must be the same, as in the `new()` method in the code section. The `<UNIT>`, `<CALCULATIONRULE>`, and the `<DESCRIPTION>` element contain arbitrary text. If that text contains '&' or '<' characters they need to be escaped (`&` or `<`) or the entire contents of the tag must be enclosed in `<![CDATA[…]]>` as shown in the `<DESCRIPTION>` tag.

The data-type of the Custom-Operation may be one of the following: `sin32, uint32, double, bool`. The parameter list must show every parameter with the correct type.

The parameter-type may be one of the following: `VAR, INT, FLOAT, BOOL, TIME`. These types are used by the UI to handle the signal-selection buttons and to check the user-input. `VAR` represents an input signal, `INT, FLOAT, BOOL` represent various constant input parameters; `TIME` represents the measurement time.

```
1;
```

The module must return true, so use `1;` as the last statement in the file.

2.4 Advanced Custom Operation (With Internal State)

The following module Gradient is a more sophisticated example, because it uses a configurable number of history data. The module computes the gradient according to the following calculation rule:

gradient[i] = input[i] - input[i - n] / time[i] - time[i - n]

Here we have the special case that we need to know the input-signal as well as the time values, which becomes important when the time values are not equally spaced. Also for every run we need to know the last values of the previous run, in order to compute the gradient of the first measure point. The parameter n (count) determines the number of measure points, between which the gradient is to be computed. This equals the number of measure points, which have to be stored between successive runs.

Here we focus on the enhanced features and do not explain the basic features once more.

```
package Gradient;
use base qw(CustomCalculationBase);

# $Id: Gradient.pm 144657 2009-08-03 18:05:51Z mibosch $

use PDL;
use PDL::Types;
use PDL::Func;
use PDL::Math;
use warnings;
use strict;

sub new
{
    return (bless {
        _signals => ['time', 'input'],
        _parameters => ['count']
        })->create(@_);
}
```

This is the standard header and `new()` function.

```
sub calc
{
    my ($this, $dim, $time, $input, $count) = @_;
    return undef if $count < 1;
    my $timedelay = $this->delay('timestate', $time);
    my $inputdelay = $this->delay('inputstate', $input);
    return ($input-$inputdelay) / ($time-$timedelay);
}
```

The gradient at sample number i is defined as the difference of values divided by the difference of times:

gradient[i] = input[i] - input[i - count] / time[i] - time[i - count]

This means in addition to the current input values and time we also need previous samples, possibly from a previous call to `calc()`. This is achieved by the `CustomCalculationBase::delay()` function. The first parameter to the function is the name of the member variable that is used to store the state between calls to `calc()`. The second parameter is the input signal that is to be delayed. The amount of delay is specified during the initialization phase which we will see next.

```
sub init
{
    my ($this) = @_;
    my $count = $this->{count};
    return undef if $count < 1;
    $this->initdelay('timestate', zeroes($count));
    $this->initdelay('inputstate', zeroes($count));
    return 0;
}
```

The `init()` method will be called whenever a measurement is started. Its purpose is to initialize all object members that store the state between calls to `calc()`. The return value has no meaning but must be 0.

For each use of `delay()` in the `calc()` method we need to initialize the corresponding delay state here. This is done using the `initdelay()` method. The first parameter is again the name of the state variable with must match the corresponding call to `delay()`. The second parameter specifies by how many samples the signal should be delayed and which values `delay()` should return until the first delayed sample is available.

```
=pod

<CUSTOMLIST>

  <VERSION>2.0</VERSION>

  <CUSTOM>
    <FUNCTIONNAME>Gradient</FUNCTIONNAME>
    <UNIT>N/A</UNIT>
    <DATATYPE>double</DATATYPE>
    <CALCULATIONRULE>Gradient::new(${time}, ${input}, ${count});
    </CALCULATIONRULE>
    <DESCRIPTION><![CDATA[first derivative of last <count> samples:
  <input>(k) - <input>(k - <count>)) / (t(k) - t(k - <count>))
  k is the measure sample at current <time>
parameters:
  <time>  = measure time
  <input> = input signal
  <count> = number of samples]]>
    </DESCRIPTION>
    <FLAGS>History</FLAGS>
    <PARAMETERLIST>
      <PARAMETER><NAME>${time}</NAME><VALUE>${'~~measureTime~~'}</VALUE>
                 <TYPE>VAR</TYPE></PARAMETER>
      <PARAMETER><NAME>${input}</NAME><TYPE>VAR</TYPE></PARAMETER>
      <PARAMETER><NAME>${count}</NAME><TYPE>INT</TYPE></PARAMETER>
    </PARAMETERLIST>
  </CUSTOM>

</CUSTOMLIST>

=cut


1;
```

The XML section is mostly the same as in the previous example. There are however two noteworthy differences:

The <FLAGS> tag is now set to `History` (whereas earlier it was `NoHistory`). The History flag notifies the system that the custom operation has state variables that need to be preserved between calls to calc(). It is important to set this flag to avoid performance optimizations which may lead to faulty results.

The first parameter of the custom calculation (`$time`) is hardwired to the special signal `${'~~measureTime~~'}` by using the <VALUE> tag. The measure time signal contains the timestamps corresponding to the input samples. Note: Timestamps are the same for all input signals. This is enforced by MDA and INCA by first interpolating all input signals to one set of timestamps as chosen by the user in the Calculated Signals dialog (periodic or same as signal).

2.5 Writing and Debugging Custom Operations

To get you up and running with your own custom calculation the directory

<EtasShared>`\CalculatedSignals\PM\Templates`

contains two sample custom calculations that can be easily adapted to your needs:

- SampleSimple: custom calculations without state
- SampleHistory: custom calculations with state

Both templates contain at their start a list of steps you need to follow to create your own custom calculation module.

When debugging your custom calculation the first step should always be to try to run your code directly with Perl, before you try it in MDA or INCA. Use the Perl interpreter provided by the ETAS product you are targeting. The Perl interpreter can be found at

<EtasShared>\perl586\bin\perl.exe

of the EtasShared corresponding to the product. Just call `perl.exe` with your custom calculation module as the only argument from a command window, e.g.:

```
C:\etas\MDA7.2\ETASShared12\perl586\bin\perl.exe C:\ETASData\
MDA7.2\CalculatedSignals\PM\MyCalc.pm
```

If you get no errors you can be sure that the module will load and compile without problems.

After that you can proceed to using it in INCA or MDA. Remember however to restart INCA or MDA whenever you make changes to the Perl module, as the modules are only loaded at startup of MDA or first opening of an experiment in INCA.

Note

Make sure, that the editor you use to edit the Perl module does not create backup files in the same directory as the Perl module. MDA and INCA will read all files regardless of their file extension. Since a backup file mostly has the same content as the original this will lead to doubly defined packages and methods causing the module loading to fail.

2.6 Old-Style Custom Operations

There are still existing custom operations that do not use the `CustomCalculationBase` base class. We will call those operations old-style. An old-style custom operation has to always implement all of the following methods:

- `$this = new(@inputs)`
- `$this->reinit(@inputs)`
- `$this->init()`
- `$this->run($dim)`

Old-style custom operations have to additionally implement the following tasks which have now been taken over by the base class:

- The input signals and parameters have to be copied from the `@inputs` list to individual member variables of the custom operation by `new()` and `reinit()`. This has been taken over by the `create()` and `reinit()` methods of the base class.
- The `run()` method needs to extract the input signals and parameters from the custom operations member variables and shorten the signal data to `$dim` samples by cutting of the unused samples at the end. The result value has to be extended again to the full length of the input signals by adding dummy samples at the end. Both tasks have been taken over by the `run()` method of the base class which calls the `calc()` method to do the actual calculation work.
- The `init()` method needs to be present even if nothing is to be done. This has been token over by the default `init()` implementation on the base class which can be overridden if necessary.
- The `DESTROY()` method needed to be implemented to clean up all member data. This is now taken over by the base class.

3 Custom Operation Call Patterns

The order in which the methods of your Perl object are called and when they are called is mostly up to the caller, i.e. INCA or MDA. The only guarantee is that the first call to the object is `new()` and the last call is `DESTROY()`. Thus the general sequence of calls is as follows:

- One to many repetitions of the cycle
  - `new()`
  - any number of calls to `run()`, `init()` and `reinit()`
  - `DESTROY()`

For a definition of the methods being called see section 2.6 Old-Style Custom Operations.

To give you an idea how your Perl object is being used the following sections will present specific sequences for some example use cases. The sequences are only to be used for reference and may change in future releases of the calculated signals.

3.1 INCA (TargetServer)

For INCA the calculation will be run by the TargetServer process which runs in real time priority class. For each calculated signal defined by the user, TargetServer will create two instances of the Perl object: One for recording data and one for display data. Under normal conditions both will be working on the same input data. For an overload condition however the recording data will start to lag behind due to buffering in the acquisition devices. The display data in contrast will be reduced to a manageable rate to avoid the lag.

For both methods the TargetServer will run the calculations at its internal polling rate. If there are more than 100 samples it will break the computation into pieces of 100 samples at a time.

- Start Measurement
  - `new()`
  - `init()`
- TargetServer Polling Cycle
  - Repeat `run(100)` (until at most 100 samples left)
  - `run(remainder)`
- Stop Measurement
  - `DESTROY()`

The number of samples actually processed during each polling cycle depends on the inputs of the calculation. The calculation can only be run up to the time where all inputs have input data available. The availability of input data usually is outside the control of the Perl object running the calculation: It may for example depend on how often an Asap1b device provides data or the presence of an overload condition.

3.2 MDA (Display in Table)

MDA calculates the calculated signals whenever it needs to display data for the calculated signal. In the case of a calculated signal displayed in a table, the sequence will look as follows:

- On Display Update
  - `new()`
  - `run()` (for samples before first visible sample, only called if History flag is set)
  - `reinit()`
  - `run()` (for visible sample)
  - `DESTROY()`

3.3 MDA (Generate Measure File)

As a special case calculated signals are also calculated when the user asks MDA to generate a measurement file. When generating measurement files MDA will break down the calculation into pieces of 1000 samples as follows:

- While there are more than 1000 samples
  - `new()`
  - `run(1000)`
  - `DESTROY()`
- For remaining samples
  - `new()`
  - `run(remainder)`
  - `DESTROY()`

4 Lookup Table Generator

A lookup table is a data structure used to approximate an arbitrary function of *n* inputs and one output. The number of inputs is also called the dimension of the lookup table. Depending on the dimension a lookup tables is also called a curve (*n*=1), map (*n*=2) or cuboid (*n*=3).

The approximation is usually calculated by the control algorithm of an ECU. If you want to reproduce the calculation as a calculated signal in INCA or MDA, the lookup table generator can create a custom operation for you based on lookup table data you provide.

Note

At the moment the lookup table generator only supports curves and maps.

4.1 Using the Lookup Table Generator

The lookup table generator is located in the directory

<EtasShared>`\CalculatedSignals\PM\LookupGenerator`

The directory contains the calibration data in the form of `*.cal` files and the generator program (LookupGenerator.bat, LookupGenerator.pm).

Running the LookupGenerator.bat will read all `*.cal` files in the directory and create one custom calculation module per `*.cal` file. The resulting custom calculation modules (`*.pm` files) will be created ready to use in the directory *<EtasShared>*`\CalculatedSignals\PM`, i.e. the parent directory.

Follow these steps if you want to use lookup tables:

1. Get the data for a lookup table and put it into a new `*.cal` file in the `LookupGenerator` directory. Use the name of the lookup table as the name of the file and append the `.cal` suffix. The name may however only contain letters and digits and must start with a letter.
2. Repeat step 1 for as many lookup tables as you need.
3. Remove the `*.cal` files for lookup tables you no longer need.
4. Now run `LookupGenerator.bat`.
5. Use the generated custom operation in MDA or INCA calculated signal. The tool has to be restarted to discover the new or changed custom calculations.

The directory already contains some sample *.cal files that you can use for your first steps.

Note

*All *.cal files as well as the generated custom operations will be deleted by both installation and de-installation of the product.*

Note

Do not rename or modify the generated custom operations (.pm files), unless you have read and understood the complete chapter 2 Custom Operations.*

4.2 Using Calibration Data from INCA

INCA can provide real calibration data for any lookup table in a format suitable for the *.cal files using the following steps:

1. Open an experiment in INCA that contains the desired lookup table.
2. Select the desired lookup table and assign it to a table editor.
3. Select the widget containing the lookup table.
4. Switch to the desired page (reference or working).
5. Switch to the desired mode (physical or hex).
6. Copy the calibration data to the clipboard using the Edit->Copy all data to clipboard menu.
7. Open notepad (or any other editor that can save as ASCII).
8. Paste the data from the clipboard.
9. Save the file into the `LookupGenerator` directory, choosing a name that identifies the lookup table.
10. In Windows Explorer rename the file extension from `*.txt` to `*.cal`

Now you are ready to run the generator as explained in the previous section.

4.3 Using Calibration Data from Excel

You can also create *.cal files using Excel. You just need to arrange the data as shown in the following examples and save it according to the instructions below.

Example for a Curve:

float x	1	1,7	2,4	4,5	7
w	17	18	3	-15	-19

The areas of the Excel sheet are color coded for this manual:

- Yellow: this cell must contain the string ”float x” with exactly one space character and is used to identify a one dimensional lookup table.
- Green: contains the data for the x-axis.
- Light blue: contains the data for the curve values.

Example for a Map:

float y \ x	1	1.4	1.6	1.8	2
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

The areas of the Excel sheet are color coded for this manual:

- Yellow: this cell must contain the string ”float y \ x” with exactly three space characters and is used to identify a two dimensional lookup table.
- Green: contains the data for the x-axis.
- Light blue: contains the data for the curve values.
- Dark blue: contains the data for the y-axis.

Use the following steps to store your data from Excel to a *.cal file:

1. Arrange your calibration data for one lookup table in one Excel sheet . You may use "," or "." as the decimal separator, i.e. "5.5" is treated the same as "5,5".
2. Select "Save As…".
3. In the file selection dialog choose "Text (Tab delimited) (*.txt)" as the "Save as type:".
4. Save the file into the `LookupGenerator` directory, choosing a name that identifies the lookup table.
5. In Windows Explorer rename the file extension from `*.txt` to `*.cal`

5 Constants

In order to ease the use of constant values, a special type of Custom Operations has been introduced. It is also implemented as a Perl module and the syntax is quite similar to the Custom Operations. It is best explained when looking at one of the example modules.

5.1 Constant Custom Operation

These modules have the same header as a Custom Operation. The package name is mandatory. Include other modules with the use-statement as needed. The code section contains the functions, which return the constant value. This value is not restricted to a constant, it may be something dynamically calculated like the current date.

```
package CustomConstants;

# $Id: CustomConstants.pm,v 1.6 2002/10/30 Exp $

use warnings;
use strict;

sub main::BIRTHDAY () { 14101957.0; }
sub main::EPOCH () { time; }

=pod

<CONSTANTLIST>

   <CONSTANT>
          <NAME>BIRTHDAY</NAME>
          <COMMENT>birthday [ddmmyyyy]</COMMENT>
          <VALUE>14101957</VALUE>
   </CONSTANT>

   <CONSTANT>
          <NAME>EPOCH</NAME>
          <COMMENT>seconds since
          00:00 January 1, 1970 GMT</COMMENT>
          <VALUE>dynamic</VALUE>
   </CONSTANT>

</CONSTANTLIST>

=cut

1;
```

The comment section contains an XML-description of the constants. Each constant is identified by its name, a comment, and a value. If the value is dynamically calculated, the value field contains the word `dynamic`. The name of the constant must be identical to the name of the function in the code section. In order to specify such a constant just by its name without the package name, the name of the function needs to be prepended with the package identifier `::main`. This puts the constant name in the top-level namespace.

6 ETAS Contact Addresses

ETAS HQ

ETAS GmbH
Borsigstraße 24 Phone: +49 711 3425-0
70469 Stuttgart Fax: +49 711 3425-2106
Germany WWW: www.etas.com

ETAS Subsidiaries and Technical Support

For details of your local sales office as well as your local technical support team and product hotlines, take a look at the ETAS website:

ETAS subsidiaries WWW: www.etas.com/en/contact.php
ETAS technical support WWW: www.etas.com/en/hotlines.php