MathScript Documentation

For use with BlueDAQ from version 1.48 (MathScript version 1)

Changes		
Version	**Status**	**By**
1.00	Initial Release	SW

Contents

Introduction

The MathScript is a file with descriptions of calculations with which MathChannels can be generated. These are software channels that can be displayed and recorded in BlueDAQ like all other channels.

You can use it to convert input values, which are usually real hardware channels; but also generate signals, i.e. functions of time f(t). For the latter purpose, MathScript can also be operated without connected hardware. This is useful, for example, to test calculations with defined input signals.

Usage

If MathScript is to be used with connected hardware (measuring amplifier) at the same time, all hardware channels must first be opened with Add Channel. Then the device type "MathScript" is selected in the Add Channel dialog. A file selection dialog opens with which the desired script can be selected. Then select the result channels of the script under "Input Channel" (= left-hand variables x <no>, see below) that are to be displayed in BlueDAQ. You can select individual ones or ranges or all of them. With MathChannels , the Add Channel dialog is confirmed, too.
BlueDAQ then evaluates the script. If an error occurs, this is reported, otherwise the script is translated into an optimized internal format that is used during the measurement runtime.
If all MathChannels could be integrated correctly, they can be used like other channels; however, less parameterization is available, so that some buttons are grayed out in the configuration tab.

The channel configuration can be saved with Save Session and restored later with Load Session; It is important that the MathScript with the same file name remains in the same path (because only this is saved in the session file, but not the script content itself).

During runtime, the script runs once for each individual measured value of all channels.

Any unit can be assigned to the MathChannels; this is saved in the session file, i.e. restored with Load Session. As with all channels, the unit is selected using the drop-down menu in the Configuration tab or via the Menu bar →Channel →Unit.

If required, channels from measuring amplifiers can be hidden in the view (Channel →Hide). This is useful when this channel is an input value for MathScript but is not to be displayed.

Only one MathScript can be used, i.e. it must contain the calculation of all required MathChannels.

Creating the MathScripts

1. Form

MathScript is a text file with the extension txt.
The character coding is ASCII only, i.e. it must not contain any special characters that are not part of the ASCII character set (with the exception of the comment texts, which are ignored by the software).

Each script consists at least of instructions (at least one) and optionally of comments and global parameters. Instructions and global parameters are case-sensitive.

The end-of-line character can be in Windows or Linux form.

1.1. Instructions

Instructions are always terminated with a semicolon. Any number of spaces, tab characters and line end characters can be placed between statements (i.e. variables and keywords). More details on this in point 2.

1.2. Comments (optional)

Comments must have the # sign (cardinal / "hash tag") as the first character in a line. If they go over several lines, each line must start again with #.

1.3. Global Parameters (optional)

Global parameters have the form

_<name> = <value>

All of these identifiers therefore begin with an underscore.
They should be at the beginning of a line and have no end-of-line characters.
Currently (version 1), there are 4 different parameters:

1.3.1.

_ScriptNo = <no>

This number is only used to differentiate between different scripts for the user. The number is displayed in the Configuration tab as a serial number when a MathChannel is selected.

If the _ScriptNo entry is missing, 0 is assigned.

1.3.2.

_Version = <no>

This can be used to distinguish future extensions. If _Version is missing, version 1 is assumed.

1.3.3.

_DataFreq = <value>

The specified data rate applies, if MathScript is used without hardware, after starting the MathScript.
It denotes the number of runs per second, i.e. the number of MathChannel arrays calculated per second. The range of values is 1 to 1000 (however, high data frequencies are not generated exactly; depending on the PC performance, system load and MathScript complexity, the data frequency that can be achieved is often only approx. 300/s).

The data rate can be changed with BlueDAQ if no hardware is connected; however, this change is volatile, i.e. <value> is not automatically updated in MathScript.
If _DataFreq is missing, the default value 10 / s is used.
If measuring amplifiers are connected, the data rate of the devices applies and _DataFreq is ignored.

It is recommended (for many reasons) to set them all to the same data frequency if possible, in case of several devices are opened.

1.3.4.

_InArrayNo = <no>

If BlueDAQ also performs special sensor calculations independently of MathScript, e.g. stress measurement for rosette strain gauges or for multi-axis sensors, you can use this to specify whether the input values in ch<no> represent these already calculated (physical) values or raw values. For rosettes the raw values are strains in the unit µm/m and for multi-axis sensors bridge excitation in mV/V.

With <no> = 0 it is indicated that the input values are calculated values (default) or raw values with <no> = 1.

If the parameter _InArrayNo is missing, 0 is assumed, i.e. calculated values.
If no software-calculated special sensors are used (e.g. also with BX6 / BX8 and hardware-calculated six-axis sensor), this parameter has no meaning.

2. Creating the instructions

Instructions are categorized into pure arithmetic expressions, definitions of constants and programming instructions, and they can contain operators, keywords, variables and numeric constants. Variables designate input and output values.

2.1. Arithmetic expressions

Non-conditional arithmetic expressions always have the form

x<no>= <expression>;

x <no> is the specified form of the left-hand variable; <no> goes from 1 to 99, i.e. up to 99 expressions can be defined, see 2.2.2.

The <expression> itself may also contain left-hand variables, provided that these have been defined before. It contains variables (see 2.2.), numeric constants (see 2.3.) and operators. Operators are those of the basic arithmetic operations and arithmetic functions (see appendix). All arithmetic functions have exactly one argument. The basic operators are used in normal mathematical forward order. They include:

+	<Addition>
-	<Subtraction>
*	<Multiplication>
/	<Division>
^	<Power-of>

Terms in the expression can be structured with round brackets. The preference of multiplication and division over addition and subtraction is known to MathScript; However, the use of brackets is recommended for very complex expressions.

All calculations are carried out internally using the 64-bit double data format.

Example 1:

```
# Multiply the values of channel 1 (from hardware) by 0.3 and add 5 to the product and assign the
#result to the variable x1:
x1= 5 + ch1*0,3;
```

The left-hand variables are to be used with all their right-hand expressions (including conditional ones) in ascending order, i.e. the first is x1, then x2 follows (if present) etc.

2.2. Variables

Variables have a fixed notation with a fixed meaning. The total number of variables used is limited to 280; for further restrictions see chapter 3.

2.2.1. Current hardware input values

They designate measured values of the current run. The notation is:

ch<no> ch1 … ch99

<no> refers to the channel number in BlueDAQ, which is displayed in "Actual Channel" and can range from 1 to 99. The input channel must exist, i.e. it must have been opened before with Add Channel or be part of the session file. Input values may only be used right-handed, i.e. only read.

2.2.2. Output values

They indicate left-hand results of calculations. They can therefore be opened with Add Channel, but do not have to, i.e. they can also be intermediate values that are not displayed. Defined values may also be used on the right in the following expressions, i.e. written and read. The notation is:

x<no> x1 … x99

<no> is a sequential number, starting with 1, then 2, and so on, up to 99.

Example 2:

```
x1= ch1*0,3 + 5;
# Calculate ch7 to the power of ch1 * 0.3 + 5 and assign the result to x2
x2= ch7^x1;
```

2.2.3. Input values from the last run

With each calculation run, i.e. at each point in time at which the array of input channel values is acquired, MathScript executes once and the input values ch <no> are saved. The input values from the previous run can be accessed with:

ch<no>L ch1L … ch99L

The input values of the last run can also only be used as right-hand values, i.e. only read.

Example 3:

```
# Form the discrete slope, i.e. the difference to the last measured value of channel 1 and
# assign this result (the temporally uncorrelated differential dx) to the variable x1:
x1= ch1 – ch1L;
```

2.2.4. Output values from the last run

With each calculation run, also the output values x<no> are stored. The output values from the last run can be accessed with

x<no>L x1L … x99L

Also the ouput values of the last run can also only be used as right-hand values, i.e. only read.. <no> corresponds to <no> of the x-value, e.g. x1L is the last value of x1. That's why x<no> must be defined in the script before.

Examample 4:

Create a sine signal (on the maening of CNT: see 2.2.5.)
x1= sin(CNT*6,2832/100);
Calculate the deviation of x1, i.e. a cosine signal with the same magnitude of 1:
x2= (100/6,2832) * (x1-x1L);

2.2.5. Special values

Until now (version 1) there are two special variables:

CNT and FREQ.

CNT is a running integer counter, that is initialized to 0 at a Reset-Event (see 3.) and incremented by 1 in every run after the execution of all instructions. It can be reset programmatically by using the RESET instruction (see 2.3.4). If that is never done it will reset itself to zero after 4.294.967.295 (2^32 -1) runs.
CNT is, among other things, suitable for generating functions f (t) that are dependent on the discrete time, i.e. signals with a discretized time, because the time between two runs corresponds to 1 / FREQ, i.e. the measurement data period (mathematically, such a signal is calculated with y = f (kT), where k = CNT and T = 1 / FREQ). For processing, CNT is converted internally into a floating point number.

See examples 4 and 6.

FREQ is the currently set measurement data frequency (total sampling rate) of the system. If there are several measuring amplifiers with different measurement data frequencies, FREQ is the mean value of the maximum and minimum measurement data rate (this special case is not recommended, however).

Example 5:

Form the integrals of channel 1 with respect to time over each measurement data period
x1= ch1 / FREQ;

2.3. Numerical Constants (Literals)

Constants can be written as an integer, as a decimal number with decimal places or as a (scientific) exponential representation on base 10. The decimal separator usually corresponds to the language set in the Windows OS. If this is a comma (e.g. with German Windows), a period may exceptionally be used, too. Literals can be used in expressions or programming instructions, but have to be present in the

definition of constants, right beneath the equal sign =.
Examples:
48 3.1415926 -5 2.598E5 -1.586E-3

2.4. Definition of Constants

Substitute identifiers can be defined for numeric constants / literals. These may be used in following expressions or programming instructions. The definition has the form:

const<String>=<numeric constant>;

<String> stands directly after const and may contain all uppercase and lowercase letters and digits, but no special characters such as _ ? . , * - <space>.

Example 6:

```
constPI=3,1415926;
constE= 73000;
conste = 2.718281828;
const23 = -41.7E-6;
x1= ch1*constE / (constPI * (1+const23*conste));
```

In example 6, x1 with ch1 = 1 gives the result 23239.2559. Note that *conste* and *constE* are treated as different (case-sensitive).

2.5 Programming instructions

All programming keywords are written in uppercase letters. Programming statements must ALWAYS be terminated with a semicolon. The notation of the variables (see 2.2.) is the same as for arithmetic expressions.
There are 2 categories of instructions: iterations and buffer instructions.

2.5.1. Iterations

Iterations are statements that are only executed when a condition is met. They can consist of a single IF statement or an IF / ELIF / ENDIF block.
IF statements have the form:

IF(<condition>) { <instruction>};

IF/ELIF/ENDIF blocks have the form:

IF(<condition>) { <instruction> };
ELIF(<condition>) { <instruction> };
<optionally other ELIF statements>

ENDIF;

The Condition

Only one condition can be checked in each IF / ELIF instruction, so a Boolean combination of several conditions is not possible within one instruction. The condition consists of a variable and a numeric constant or of two variables that are compared with each other. The following 6 comparison operators exist:

>	<greater than>
<	<less than>
>=	<greater or equal>
<=	<less or equal>
=	<equal>[1]
!=	<not equal>

The condition in an ELIF instruction is only checked if all previous IF and ELIF conditions are not met. As soon as a condition is met in an IF / ELIF / ENDIF block, the corresponding statement is executed and all subsequent ELIF conditions are no longer checked (ELIF corresponds, for example, to the else if () behavior in the C language).
If none of the conditions apply, no instruction is executed and the left-hand value in the instruction x <no> retains the last assigned value, i.e. also that of the previous script run, if it was not yet assigned in the current run.

Tip: In order to accomplish the behavior of a typical ELSE instruction, i.e. an instruction that is executed if none of the previous IF / ELIF conditions apply, the immediately preceding IF / ELIF condition can be logically reversed with the same arguments, see example 10.

The Instruction in IF / ELIF

The instruction is enclosed in curly brackets. With the exception of the RESET command (see 2.5.2), instructions consist of exactly one assignment of the form

x<no>= <expression>

The expression can be exactly one arithmetic instruction or one buffer instruction.

Within an IF / ELIF / ENDIF block, all x <no> variables on the left must be the same (version 1).

Example 7:

```
# Script without hardware, creates a triangle signal with the amplitude constA= 5
# The signal period is divided into 3 sections, i.e. defined section by section
constA = 5;
```

1 The equal / unequal comparison takes place in the interval of the so-called machine epsilon of the 64-bit double mantissa

```
IF(CNT < 25) {x1= CNT*4*constA/100};
ELIF(CNT < 75) {x1= 2*constA-(CNT*4*constA/100)};
ELIF(CNT < 100) {x1= (CNT*4*constA/100) - 4*constA};
ENDIF;
# Resettimg the counter. Done after the last ELIF statement in the block above,
# so that CNT has a value range from 0 to 99
IF(CNT=99) {RESET(0)};
```

2.5.2. Buffer Instructions

Buffer instructions enable summing up or averaging of values that follow one another in time. Internal accumulation registers are used to store the totals. Up to 23 accumulation registers can be used.

SUM Instruction

The SUM instruction is used to sum values of the same variable that are consecutive in time. It has the form:

x<no>= SUM<a>(<arg>, N);

Whereby:

x<no>: Result variable, left-handed value

<a>: Number of the accumulating register. The value range is from 1 to 23. If <a> is omitted, if will be substituted by 1. With this, e.g. different arguments (variables) can be summed up independently of each other. A new <a> must then be selected for each SUM instruction, in ascending order. The MEAN instruction uses the same register bank, that's why, when selecting the number for <a>, the MEAN instructions have to be counted, too.

<arg>: Argument, i.e. variable, that shall be summed up. Possible arguments are: ch<no>, x<no> (if defined before), ch<no>L, x<no>L (if x<no> defined before).

N: Number of values to be added. If N> 1, each result is the sum of the (N-1) -last values plus the current value1 in <arg>. However, this only applies if N or more values are already available after a RESET event. Otherwise the result is the sum of all existing values in <arg>.

If N = 0, the system continues to add. The result is the sum of all values in <arg> from the last RESET event (see below) or RESET command to the next RESET event / command.

Example 8:

```
# Form the sum of the current and the last 9 values of channel 1
x1= SUM1(ch1, 10);
# Form the sum of the current and the last 99 values of channel 7
x2= SUM2(ch7, 100);
```

MEAN Instruction

The MEAN instruction is used to calculate the arithmetic mean value of consecutive values of the same variable. It has the form:

x<no>= MEAN<a>(<arg>,N);

The meaning of the placeholders no, a, arg and N is the same as for the SUM statement. If you select <a>, it starts with 1 and then counts on, including the SUM instructions.
The difference to the SUM instruction is that with MEAN the sum is always divided by the number of added values. If N> 1, this is N if there are already N values since the RESET event; otherwise the number of existing (= totaled) values. If N = 0, it is divided by the number of accumulated values and the divisor becomes larger and larger - up to a new RESET event.
This results in a moving average.

Example 9:

```
# Form the mean value over the current and the last 9 values of channel 1
x1= MEAN1(ch1, 10);
# Form the mean value over the current and the last 99 values of channel 7
x2= MEAN2(ch7, 100);
```

RESET Instruction

The RESET command is used in the execution part of a conditional statement. This can be used to reset the general counter "CNT" or one of the sum registers to 0.

Form:

IF(<Condition>) { RESET(<a>) };

The parameter <a> specifies what is to be reset:

a=0: General counter CNT.

A= 1..23: Accumulation register; according to the parameter <a> of a previously used SUM(<a>) or MEAN(<a>) instruction.

See example 7:

IF(CNT=99) {RESET(0)};

CNT thus takes values from 0 to 99.

Example 10:

```
# Calculate x2= integral of ch1 within the boundary y >10^-6
# Outside this boundary (i.e. y<=10^-6), the last value of x2 is output repeatedly
x1=ch1/FREQ;
```

```
IF(ch1>1E-6){x2=SUM(x1, 0)};
ELIF(ch1<=1E-6){RESET(1)};
ENDIF;
```

Example 10.1:

As Example 10, but x2 is set to 0 outside the integral boundary:

```
x1=ch1/FREQ;
IF(ch1>1E-6){x2=SUM(x1, 0)};
ELIF(ch1<=1E-6){x2= 0};
ENDIF;
IF(ch1<=1E-6){RESET(1)};
```

3. Execution of the MathScript

After the script has been opened and integrated (i.e. after clicking Connect in the Add Channel dialog or after opening a session with MathChannels) it is first evaluated ("parsed") and checked for some errors. Many types of errors can be detected, but not all. If no error is detected, the script is translated into an internal structure that helps to carry out the calculations as efficiently as possible during runtime. This can be done with a data frequency of up to 12000/s, depending on the PC performance, power availability and script complexity.
All runtime data (e.g. sum register, last measured and output values) are set to 0 before the first execution. This Reset event is also triggered once when you click on Start Measuring or Start Recording (in the Yt and XY recorders).

The calculation at runtime takes place in 3 main steps:

A) Execution of all programming instructions that do not require any current output values (x<no> variables) as inputs, i.e. as right-handed values.

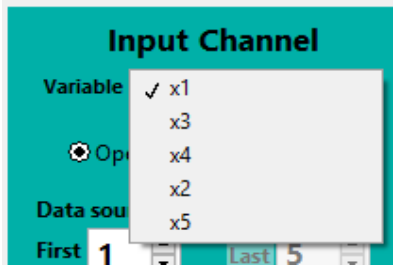
B) Executing all non-conditional arithmetic instructions

C) Execution of all programming instructions that require current output values from previous calculations (x<no> variables) as input.

This procedure has the side-effect that the instructions may be executed in a different order than they are given in the MathScript. In addition, the results of instructions that are executed in step C (i.e. that require current output values as input) cannot be processed further, i.e. their results x<no> may no longer be used as a right-hand input value.

The execution order is shown indirectly in the Add Channel dialog after the script file has been opened. On the right under Input Channel is a drop-down menu called *Variable*.

For instance, the order x1, x3, x4, x2, x5:



results from the following script (example 11):

```
x1=MEAN1(ch1,1000);
IF(CNT < 25) {x2= CNT*4*10*x1/100};
ELIF(CNT < 75) {x2= 2*10*x1-(CNT*4*10*x1/100)};
ELIF(CNT < 100) {x2= (CNT*4*10*x1/100)-(4*10*x1)};
ENDIF;
IF(CNT=99) {RESET(0)};

x3= MEAN2(ch2,3);
x4=x1/FREQ;

IF(ch2>1E-6){x5=SUM3(x4,0)};
ELIF(ch2<=1E-6){x5=0};
ENDIF;
IF(ch2<=1E-6){RESET(3)};
```

The calculation of x2, i.e. all lines 2 to 6, is shifted to step C because these instructions require the value x1 on the right-hand side. x1 (line 1) and x3 (line 7) are executed in step A and x4 (line 8) in step B, using the value x1 already calculated in A.
The calculation of x5 (lines 9 to 12) is carried out in step C because the value x4 is required on the right-hand side.

If the script in example 11 is changed in line 7 as follows:

```
x1=MEAN1(ch1,1000);
IF(CNT < 25) {x2= CNT*4*10*x1/100};
ELIF(CNT < 75) {x2= 2*10*x1-(CNT*4*10*x1/100)};
ELIF(CNT < 100) {x2= (CNT*4*10*x1/100)-(4*10*x1)};
ENDIF;
IF(CNT=99) {RESET(0)};

x3= MEAN2( x2, 3);
x4=x1/FREQ;

IF(ch2>1E-6){x5=SUM3(x4,0)};
ELIF(ch2<=1E-6){x5=0};
ENDIF;
IF(ch2<=1E-6){RESET(3)};
```

Then it cannot be carried out correctly because the calculation of x3 depends on a calculation result carried out in step C, namely x2.

The total number of variables used is also limited by the number of programming

instructions in step A, whereby IF ... ELIF ... ENDIF blocks are counted as one instruction here. The total number of different variables plus the instructions in A must be less than 287.

3.1.More tips for using MathScript

- If you activate the radio button "Open All Channels" in the Add Channel dialog, the MathChannels will be opened in the above execution order. You can also open channels individually, in any order (the only important thing is that the required hardware channels have been opened beforehand).

- In order to use a different MathScript than the one that is already integrated, all MathChannels must first be removed with Remove Channel.

- If hardware channels are required for the MathScript that should not be displayed, you can "hide" them with Channel -> Hide. This state is also saved with Save Session.

Annex

List of arithmetic Functions

Function	Name	Description
abs(*x*)	Absolute Value	Returns the absolute value of *x*.
acos(*x*)	Inverse Cosine	Computes the inverse cosine of *x* in radians.
acosh(*x*)	Inverse Hyperbolic Cosine	Computes the inverse hyperbolic cosine of *x*.
asin(*x*)	Inverse Sine	Computes the inverse sine of *x* in radians.
asinh(*x*)	Inverse Hyperbolic Sine	Computes the inverse hyperbolic sine of *x*.
atan(*x*)	Inverse Tangent	Computes the inverse tangent of *x* in radians.
atanh(*x*)	Inverse Hyperbolic Tangent	Computes the inverse hyperbolic tangent of *x*.
ceil(*x*)	Round Toward +Infinity	Rounds *x* to the next higher integer (smallest integer *x*).
cos(*x*)	Cosine	Computes the cosine of *x*, where *x* is in radians.
cosh(*x*)	Hyperbolic Cosine	Computes the hyperbolic cosine of *x*.
cot(*x*)	Cotangent	Computes the cotangent of *x* (1/tan(*x*)), where *x* is in radians.
csc(*x*)	Cosecant	Computes the cosecant of *x* (1/sin(*x*)), where *x* is in radians.
exp(*x*)	Exponential	Computes the value of *e* raised to the *x* power.
expm1(*x*)	Exponential (Arg) – 1	Computes one less than the value of *e* raised to the *x* power ((*e*^*x*) – 1).
floor(*x*)	Round To –Infinity	Truncates *x* to the next lower integer (largest integer *x*).
getexp(*x*)	Get Exponent	Returns the exponent of the input numeric value such that x = mantissa * 2^exponent.
getman(*x*)	Get Mantissa	Returns the mantissa of the input numeric value such that x = mantissa * 2^exponent.
int(*x*)	Round To Nearest	Rounds *x* to the nearest integer.
intrz(*x*)	—	Rounds *x* to the nearest integer between *x* and zero.
ln(*x*)	Natural Logarithm	Computes the natural logarithm of *x* (to the base of *e*).
lnp1(*x*)	Natural Logarithm (Arg +1)	Computes the natural logarithm of (*x* + 1).

log(*x*)	Logarithm Base 10	Computes the logarithm of *x* (to the base of 10).
log2(*x*)	Logarithm Base 2	Computes the logarithm of *x* (to the base of 2).
rand(0)	Random Number (0 – 1)	Produces a floating-point number between 0 and 1 exclusively. Remark: Must write rand(0), i.e. contain dummy argument 0.
sec(*x*)	Secant	Computes the secant of *x*, where *x* is in radians (1/cos(*x*)).
sign(*x*)	Sign	Returns 1 if *x* is greater than 0, returns 0 if *x* is equal to 0, and returns –1 if *x* is less than 0.
sin(*x*)	Sine	Computes the sine of *x*, where *x* is in radians.
sinc(*x*)	Sinc	Computes the sine of *x* divided by *x* (sin(*x*)/*x*), where *x* is in radians.
sinh(*x*)	Hyperbolic Sine	Computes the hyperbolic sine of *x*.
sqrt(*x*)	Square Root	Computes the square root of *x*.
tan(*x*)	Tangent	Computes the tangent of *x*, where *x* is in radians.
tanh(*x*)	Hyperbolic Tangent	Computes the hyperbolic tangent of *x*.
gamma(x)	Gamma function	
ci(x)	Cosine integral	
si(x)	Sine Integral	
spike(x)	Spike function	The following equation defines the spike function: spike(x) = 1, if 0 x < 1 and 0 elsewhere.