Contents

1 MeLa v1.0 reference manual

1.1 Language description

A MeLa application is composed of three main parts:

- The mission configuration to define basic parameters of the instrument.
- The coordinator to define when to execute the acquisition mode(s).
- The acquisition mode(s) to acquire and process the sensors data. There are two types of acquisition modes, the `ContinuousAcqMode` that processes data without stopping, and the `ShortAcqMode` that processes only one packet of data.

```
Mission:
    ...

Coordinator:
    ...

ContinuousAcqMode acq1:
        ...

ShortAcqMode acq2:
        ...
```

1.1.1 Mission configuration

The mission configuration allows to define the depth of the dive, and the maximum time that the instrument can pass at this depth.

```
Mission:
    ParkTime: 14400 minutes;
    ParkDepth: 1500 meters;
```

1.1.2 Coordinator

The coordinator defines when to execute an acquisition mode during the steps of a dive (i.e., descent, park and ascent). For `ShortAcqMode`, a time interval between each execution must be defined. This is not needed for `ContinuousAcqMode` that never stops during the step in which it is executed.

```
Coordinator:
    DescentAcqModes: acq1;
    ParkAcqModes: acq1, acq2 every 10 minutes;
    AscentAcqModes: acq2 every 10 minutes;
```

1.1.3 Acquisition modes

A `ContinuousAcqMode` processes data in a streamed way, without stopping. It is more adapted to monitor sporadic events (i.e., that appends from time to time), but it can use a lot of processor time, especially if the sampling of the sensor is high. A `ContinuousAcqMode` is divided in several parts:

- The `Input` part allows defining the input sensor to use and its configuration, for example it can be the hydrophone with a sampling frequency of 200 Hz.
- The `Variables` part allows defining the variables used for data processing. A list of data types is given in section 1.3.
- The `RealTimeSequence` part contains instructions to process the data in real time. This sequence is executed in a loop each time time a packet of data is sent by the sensor. It gives the guarantee that all the data will be processed, without missing a sample, such that the data cannot be truncated. Only one `RealTimeSequence` can be defined.
- The `ProcessingSequence` part is optional but can be used for instructions with an execution time too long for the `RealTimeSequence`. During the execution of this sequence some data sent by the sensor can be missed. It is possible to define several `ProcessingSequence` that can be called from the `RealTimeSequence` or from another `ProcessingSequence`.

```
ContinuousAcqMode acq1:

Input:
    HydrophoneBF(40);

Variables:
    Int i;
    ArrayFloat array(10);

RealTimeSequence seq1:
    ...
endseq;

ProcessingSequence seq2:
    ...
endseq;
```

A `ProcessingAcqMode` have a very similar structure, the only difference is that it does not contain a `RealTimeSequence` because only one packet of data is processed.

1.1.4 Instructions

Instructions are called inside the sequences of instructions (i.e., the RealTimeSequence or ProcessingSequence). The instructions can be:

- An operation, such as `c = a + b`. Only two operands are accepted in the current version of MeLa. The operands must be integers of floats. The possible operators are `+`, `-`, `*` and `/`.
- A function call, such as `mean(array, r);` which computes the mean values of the array and put the result into the `r` variable.
- A `if` condition, such `if a > 10 && b > c`. The comparison operators are `<`, `<=`, `>`, `>=`, `==`, `&&`, `||`. A condition must also contain a probability, such as `@probability = 1 per hour`. These probabilities are used by MeLa to compute the battery lifetime of the float and the amount of data transmitted each month. The possible time units are `sec`, `min`, `hour`, `day` and `week`.
- A `for` loop, such as `for i, v in array`. Each loop iteration read an element of the array from the first to the last, the index of the current element is put in the `i` variable, and its value in the `v` variable.

```
RealTimeSequence seq1:

   /* Add two variables */
   c = a + b;

   /* If variables exceed a value */
   if a > 10 && b > c:
      @probability = 1 per hour
      */ Compute the mean of the array */
      mean(array, r);
   endif;

   /* Add each element of the array to the a variable */
   for i, v in array:
      a = a + v;
   endfor;

endseq;
```

1.2 Constants

Constants can be used to set the parameters of variables or functions. There are three types of constants that are integers, floating point numbers and strings.

Integers constants are defined as real numbers such as `8467` or `-16`. They are encoded on 32 bits. The largest possible value of a 32 bits integer is 2147483647, the smallest positive is 1 and the smallest negative is -2147483648.

Floating points numbers must be defined with a fractional part such as `8467.54` or `-16.45532`. The fractional part must be defined even if it is null to be recognized as a floating point number by the language, for example `10.0`, otherwise the number will be considered as an integer. It is also possible to define floating point numbers under an exponential form such as `+6.840015400e-01`. Floating point numbers are also encoded on 32 bits. The largest possible value of a floating point number is 3.4028235×10^{38}, the smallest positive is 1.175494×10^{-38} and the smallest negative is $-3.4028235 \times 10^{38}$.

Strings are written between quotation marks, such as `"This is a string"`. MeLa does not include functions to manipulate strings, however they can be used as separator for data recorded on files.

1.3 Data types

All data types are presented with the following pattern:

```
DataType variableName;
```

or if parameters can be specified:

```
DataType variableName(parameter1, parameter2, ...);
```

1.3.1 Boolean

Boolean variables are declared as:

```
Bool b;
```

1.3.2 Integer

Integer variables are declared as:

```
Int i;
```

1.3.3 Float

Floating point variables are declared as:

```
Float f;
```

1.3.4 Complex numbers

Complex numbers contains an imaginary part and a real part, they are mainly used to compute Fast Fourier Transform. They can be either integers of floating point numbers.

```
ComplexInt ci;
ComplexFloat cf;
```

1.3.5 Arrays

Arrays can be defined for integers, floating points and complex numbers.

```
ArrayInt ai(length, initvs);
ArrayFloat af(length, initvs);
ArrayComplexInt aci(length);
ArrayComplexFloat acf(length);
```

Parameters:

1. `length` is the length of the array.
2. `initvs` are optional parameters to initialize the values of the array. The number of values must be equal to the length of the array. Arrays of complex numbers cannot be initialized in the current version of MeLa.

Examples:

```
ArrayInt ai1(3);
ArrayInt ai2(10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
ArrayFloat af1(8);
ArrayFloat af2(5, 1.0, 2.0, 3.0, 4.0, 5.0);
ArrayComplexInt aci(100);
ArrayComplexFloat acf(20);
```

1.3.6 Buffers

Buffers are similar to arrays but it is possible to append data at the end of a buffer whereas it is not for an array. If the buffer is full the older data are overwritten, this is the principle of a circular buffer in which data can be appended indefinitely.

```
BufferInt ai(length);
BufferFloat bf(length);
```

Parameter:

1. `length` is the length of the buffer.

Examples:

```
BufferInt bi(10);
BufferFloat bf(10);
```

1.3.7 FFT

This data type allows to declare variables that are necessary to compute Fast Fourier Transform (FFT). The FFT can be either integer or floating points arithmetic. The function to process the FFT is presented in section 1.5.13.

```
FFTInt x(npow2);
FFTFloat x(npow2);
```

Parameter:

1. `npow2` is the size of the FFT. It must be an integer with a power of 2 value comprised between 32 and 4096 (*i.e.*, 32, 64, 128, 256, 512, 1024, 2048 or 4096).

Examples:

```
FFTInt fi(128);
FFTFloat ff(1024);
```

1.3.8 IIR

this data type allows to declare variables necessary for integer Infinite Impulse Response filters (IIR). The IIR can be either integer or floating points arithmetic.

```
IIRInt x(norder, dorder, ncoeffs, dcoeffs);
IIRFloat x(norder, dorder, ncoeffs, dcoeffs);
```

Parameters:

1. `norder` is the number of coefficients at the numerator.
2. `dorder` is the number of coefficients at the denominator.
3. `ncoeffs` are the coefficients of the numerator, the number of values must be equal to the numerator number of coefficients.
4. `dcoeffs` are the coefficients of the denominator, the number of values must be equal to the denominator number of coefficients.

Examples:

```
IIRInt ii(2, 5, 1, -2, 1, -2, 3, -4, 5);
```

corresponds to the filter $H(z) = \frac{y(z)}{x(z)} = \frac{1-2z^{-1}}{1-2z^{-1}+3z^{-2}-4z^{-3}+5z^{-4}}$

```
IIRFloat if(2, 5, 1.0, -0.2, 1.0, -0.2, 0.3, -0.4, 0.5);
```

corresponds to the filter $H(z) = \frac{y(z)}{x(z)} = \frac{1-0.2z^{-1}}{1-0.2z^{-1}+0.3z^{-2}-0.4z^{-3}+0.5z^{-4}}$

1.3.9 FIR

Finite Impulse Response filters (FIR) are defined as IIR but do not have denominator.

```
FIRInt x(norder, ncoeffs);
FIRFloat x(norder, ncoeffs);
```

Parameters:

1. `norder` is the number of coefficients.
2. `ncoeffs` are the coefficients, the number of values must be equal to the number of coefficients.

Examples:

```
FIRInt fi(2, 1, -2);
FIRFloat ff(2, 0.1, -0.2);
```

corresponds to the filters $H(z) = \frac{y(z)}{x(z)} = 1 - 2z^{-1}$ and $H(z) = 0.1 - 0.2z^{-1}$

1.3.10 CDF24

This datatype allows to declare variables necessary to compute a CFD24. The CDF24 is a specific implementation of a wavelet transform originally used in the Mermaid algorithm[1]. A wavelet transform is basically equivalent to a bank of band pass filter.

```
CDF24Int x(nscales, nsamples);
CDF24Float x(nscales, nsamples);
```

Parameters:

1. `nscales` is the number of scales of the transform (*i.e.*, number of frequency bands).
2. `nsamples` is the number of samples to process.

Examples:

```
CDF24Int cdi(6, 12000);
CDF24Float cdf(6, 12000);
```

[1] Sukhovich et al., (2011), "Automatic discrimination of underwater acoustic signalsgenerated by teleseismic P-waves: A probabilistic approach", Geophysical Research Letters

1.3.11 STALTA

This datatype allows to declare variables necessary for the Short Term Average over Long Term Average (STA/LTA). The STA/LTA algorithm is commonly found for seismic processing.

```
StaLtaInt x(staLenght, ltaLenght, ltaDelay, scaling);
StaLtaFloat x(staLenght, ltaLenght, ltaDelay);
```

Parameters:

1. `staLenght` is the length of the short term average.
2. `ltaLenght` is the length of the long term average.
3. `ltaDelay` is a delay for the beginning of the LTA.
4. `scaling` is a scaling factor for the ratio. It allows to obtain a better precision when integer arithmetic is used and the ratio of the average is close to 1. For example, instead of a ratio passing from 0 to 1 without intermediate values, a scaling factor of 10 allows to pass from 0 to 10 with all intermediate integer values.

Examples:

```
StaLtaInt sti(400, 4000, 400, 1000);
StaLtaFloat stf(400, 4000, 400);
```

In these examples, the STA has a length of 400 samples, the LTA has a length of 4000 samples and is delayed of 400 samples (*i.e.*, the LTA average the samples between the index 400 and 4399). The ratio of the integer implementation is multiplied by 1000.

1.3.12 Trigger

This datatype allows to declare variables necessary for the trigger algorithm. The trigger is used to detect changes of a signal compared to to a threshold value. The function that processes the trigger (section 1.5.31) return a boolean value depending of the parameters defined here.

```
TriggerInt x(mode, threshold, delay, minInterval);
TriggerFloat x(mode, threshold, delay, minInterval);
```

Parameters:

1. `mode` is the mode of activation of the trigger, there are four possible modes:
   (a) `HIGH`: the trigger will return `true` if the signal is above the threshold value.
   (b) `LOW`: the trigger will return `true` if the signal is under the threshold value.

(c) `RISING_EDGE`: the trigger will return `true` if the signal pass above the threshold value, but only for the instant when it passes from a lower to a higher level.

(d) `FALLING_EDGE`: the trigger will return `true` if the signal pass under the threshold value, but only the instant when it passes from a higher to a lower level.

2. `threshold` is the threshold to compare with the signal.
3. `delay` is a delay to activate the trigger. It can be used to wait a number of `delay` samples after a signal pass over a threshold in order to have more data to process.
4. `minInterval` is the minimum sample numbers between each trigger, it can be used to ignore successive triggers close to each others.

Examples:

```
TriggerInt ti(RISING_EDGE, 2500, 4000, 10000);
TriggerFloat tf(HIGH, 2.5, 4000, 10000);
```

In the first example, the trigger will return `true` if the signal **pass** above a value of `2500`. In the second example, the trigger will return `true` if the signal **is** above a value of `2.5`.
For both examples, the trigger is always delayed of 4000 samples and there cannot be less than 10000 samples between each trigger.

1.3.13 Distribution

This type of data is used by the function that compute the cumulative distribution (section 1.5.8).

```
DistributionInt x(length, xvalues, yvalues);
DistributionFloat x(length, xvalues, yvalues);
```

Parameters:

1. `length` is the length of the distribution.
2. `xvalues` is the x-axis values, the number of values must be equal to the length of the distribution.
3. `yvalues` is the y-axis values, the number of values must be equal to the length of the distribution.

Examples:

```
DistributionInt di(5,
                   1, 2, 3, 4, 5,
                   3, 6, 9, 8, 2);
DistributionFloat df(5,
                     1.0, 1.1, 1.2, 1.3, 1.4,
                     129.0, 326.0, 446.0, 290.0, 230.0);
```

In these examples, both distributions contains 5 numbers, where the x values are given in the second line of each distribution and the y values are given in the third line of each distribution.

1.3.14 File

To declare a file for recording data. The files cannot be read. All data recorded in a file are transmitted trough satellite.

```
File f;
```

No parameters are required.

1.4 Array and Buffer functions

1.4.1 clear

Set all values of an array or a buffer to 0.

```
clear(ArrayInt array);
clear(ArrayFloat array);
clear(ArrayComplexInt array);
clear(ArrayComplexFloat array);
clear(BufferInt buffer);
clear(BufferFloat buffer);
```

Parameters:

1. `array` is the array to clear.

Example:

```
ArrayInt a(5);
put(a, 0, 56);
put(a, 1, 1896);
put(a, 2, 89);
put(a, 3, 67);
put(a, 4, 156);
clear(a);
```

In this example, some values are put in the `a` array and then it is cleared which means that the array will be filled with zero values.

1.4.2 copy

Copies the content of array1 starting from index1 and for the specified length to the array2 starting from index2.

```
copy(ArrayInt src, Int isrc,
            ArrayInt dst, Int idst, Int len);
copy(ArrayFloat src, Int isrc,
            ArrayInt dst, Int idst, Int len);
```

```
copy(ArrayComplexInt src, Int isrc,
             ArrayInt dst, Int idst, Int len);
copy(ArrayComplexFloat src, Int isrc,
             ArrayInt dst, Int idst, Int len);
```

Parameters:

1. `src` is the array to copy.
2. `isrc` is the index from which to start the copy.
3. `dst` is the array in which to realize the copy.
4. `idst` is the index from which to start the copy.
5. `len` is the length to copy.

Example:

```
ArrayInt a(10);
ArrayInt b(10);
ArrayInt c(20);
copy(a, 0, c, 0, 10);
copy(b, 5, c, 10, 5);
```

In this example, the first call to the `copy` function copies the content of `a` in the first half part of `c` and the second call to the `copy` function copies half of the content of 'b' in the second half part of `c`.

1.4.3 get

Get a value in an array at a specified index.

```
get(ArrayInt array, Int index, Int value);
get(ArrayFloat array, Int index, Float value);
get(ArrayComplexInt array, Int index, ComplexInt value);
get(ArrayComplexFloat array, Int index, ComplexFloat value);
```

Parameters:

1. `array` is the array in which to get a value.
2. `index` is the index where to get the value.
3. `value` is the variable that will contain the value.

Example:

```
ArrayFloat a(10);
Float v;
get(a, 5, v);
```

In this example, the value at index 5 of the `a` array is put into the `v` variable.

1.4.4 put

Put a value in an array at a specified index.

```
put(ArrayInt array, Int index, Int value);
put(ArrayFloat array, Int index, Float value);
put(ArrayComplexInt array, Int index, ComplexInt value);
put(ArrayComplexFloat array, Int index, ComplexFloat value);
```

Parameters:

1. `array` is the array in which to put a value.
2. `index` is the index where to put the value.
3. `value` is the value to put in the array.

Example:

```
ArrayFloat a(10);
Float v;
put(a, 5, v);
put(a, 8, 1.6);
```

In this example, an array of floating point numbers called `a` and a variable called v are declared. The value of `v` is put at index 5 of the array and a value of `1.6` is put at index 8 of the array.

1.4.5 push

Add a value to the end of the buffer.

```
push(BufferInt buffer, Int value);
push(BufferFloat buffer, Float value);
push(BufferInt buffer, ArrayInt value);
push(BufferFloat buffer, ArrayFloat value);
```

Parameters:

1. `buffer` is the buffer to fill.
2. `value` is the value used to fill the buffer (it can be an array).

Example:

```
BufferInt b(100);
ArrayInt a(10);

/* Add values of the array to the end of the buffer */
push(b, a);
```

In this example, the 10 values contained in the `a` array are put into the `b` buffer.

1.4.6 select

Select a specific portion of an array to work on. Once a portion of the array is selected all the following operation are done on the selected portion of the array, until it is unselected using the 'unselect' function.

```
select(ArrayInt array, Int index1, Int index2);
select(ArrayFloat array, Int index1, Int index2);
select(ArrayComplexInt array, Int index1, Int index2);
select(ArrayComplexFloat array, Int index1, Int index2);
```

Parameters:

1. `array` is the array to select.
2. `index1` is the index where to start the selected portion.
3. `index2` is the index where to stop the selected portion.

Example:

```
ArrayInt a(100);
select(a, 50, 69);
```

In this example, we select a portion of the `a` array between index 50 and 69 (included), the length of the selected portion is 20.

1.4.7 unselect

Cancel the effect of the select command.

```
unselect(ArrayInt array);
unselect(ArrayFloat array);
unselect(ArrayComplexInt array);
unselect(ArrayComplexFloat array);
```

Parameters:

1. `array` is the array to unselect.

Example:

```
ArrayInt a(10);
unselect(a, 2, 5);
unselect(a);
```

In this example, we first selected a portion of the `a` array and unselected it to come back to its normal size.

1.5 Math functions

1.5.1 abs

Absolute value.

```
abs(Int input, Int output);
abs(Float input, Float output);
abs(ArrayInt input, ArrayInt output);
abs(ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the input value.
2. `output` is the absolute value of the input value.

Example:

```
ArrayInt a(10);
ArrayInt b(10);

/* Compute the absolute values of a and put the result in b */
abs(a, b);
```

1.5.2 add

Add elements of an array (with a value) or add two arrays.

```
add(ArrayInt input1, ArrayInt input2, ArrayInt output);
add(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
TODO: add with integer or float
```

Parameters:

1. `input1` is the left operand.
2. `input2` is the right operand.
3. `output` is result of the addition of the two array (element by element).

Example:

```
ArrayInt a(10);
ArrayInt b(10);
ArrayInt c(10);

/* Add each element of a and b and put the result in c */
abs(a, b, c);
```

1.5.3 cdf24

Compute a CDF24 wavelet transform. Parameters of the CDF24 must be defined in the variable `cdf24v`. Input data are overwritten by the results of the transform.

```
cdf24(CDF24Int cdf24v, ArrayInt array);
cdf24(CDF24Float cdf24v, ArrayFloat array);
```

Parameters:

1. `cdf24v` is the variable which contain parameters to process the wavelet transform.
2. `array` is the array to process.

Example:

```
CDF24Int cdi(6, 12000);
ArrayInt a(12000);

/* Compute the CDF24 wavelet transform of the array a */
cdf24(cdi, a);
```

1.5.4 cdf24ScalesPower

Compute the power of each scale of the CDF24 from array1. The index1 and index2 values allow to select a subset of the CDF24. The results are put in array2 which must have a size at least equal to the number of computed scales.

```
cdf24ScalesPower(CDF24Int cdf24v, ArrayInt array1,
                 Int index1, Int index2, ArrayInt array2);
cdf24ScalesPower(CDF24Float cdf24v, ArrayFloat array1,
                 Int index1, Int index2, ArrayFloat array2);
```

Parameters:

1. `cdf24v` is the variable which contains parameters of the wavelet transform.
2. `array1` is the array containing the wavelet transform.
3. `index1` is the index at which the processing of the power must start.
4. `index2` is the index at which the processing of the power must end.
5. `array2` is the array containing the power for each scale.

Example:

```
CDF24Int cdi(6, 12000);
ArrayInt a(12000);
ArrayInt r(6);

/* Compute the power on the first half part of the signal */
cdf24(cdi, a);
cdf24ScalesPower(cdi, a, 0, 5999, r);
```

In this example, the CDF24 wavelet of the `a` array is processed, then the power of each scale contained in the `a` array is computed and the result is put in the `r` array. Only the first half part of the signal (from index 0 to index 5999) is used to compute the power.

1.5.5 conv

Convolution between two array.

```
conv(ArrayInt input1, ArrayInt input2, ArrayInt output);
conv(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters:

1. `input1` is the left operand.
2. `input2` is the right operand.
3. `output` is the result of the convolution between `input1` and `input2`.

Example:

```
ArrayInt a(100);
ArrayInt b(100);
ArrayInt c(100);

/* Compute the convolution of array a and b
   and put the result in c */
conv(a, b, c);
```

1.5.6 corr

Correlation between two array.

```
corr(ArrayInt input1, ArrayInt input2, ArrayInt output);
corr(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters:

1. `input1` is the left operand.
2. `input2` is the right operand.
3. `output` is the result of the correlation between `input1` and `input2`.

Example:

```
ArrayInt a(100);
ArrayInt b(100);
ArrayInt c(100);

/* Compute the correlation of array a and b
   and put the result in c */
conv(a, b, c);
```

1.5.7 cos

Cosine.

```
cos(ArrayInt input, ArrayInt output);
cos(ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the input array.
2. `input` is the result of the cosine operation.

Example:

```
ArrayInt a(100);
ArrayInt b(100);

/* Compute the cosine of array a
   and put the result in b */
cos(a, b);
```

1.5.8 cumulativeDistribution

Compute the cumulative distribution. The distribution must be defined when declaring the variable `distributionv`. This functions sum the y-values of the distribution until to reach the limit that is compared to the x-values of the distribution.

```
cumulativeDistribution(DistributionInt distributionv,
                       Int limit, Int result);
cumulativeDistribution(DistributionFloat distributionv,
                       Float limit, Float result);
```

Parameters:

1. `distributionv` is the distribution variable (section 1.3.13).
2. `limit` is the limit until which is computed the cumulative distribution.
3. `result` is the result of the cumulative distribution.

Example:

```
DistributionInt dist(100);
Int lim;
Int res;

/* Compute the cumulative distribution until
   to reach the lim value */
cumulativeDistribution(dist, lim, res);
```

1.5.9 diff

Forward finite difference (derivative approximation). The equation used for this algorithm is $output[i] = input[i+1] - input[i]$ and for the last element of the array $output[i] = input[i]$.

```
diff(ArrayInt input, ArrayInt output);
diff(ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the input array.
2. `input` is the result of the forward finite difference.

Example:

```
ArrayInt data(100);
ArrayInt res(100);

/* Compute the forward finite difference
   of data and put the result in res */
diff(data, res);
```

1.5.10 div

Divide elements of an array with a value, $a[i]/v$, or divide two arrays element by element, $a1[i]/a2[i]$.

```
div(ArrayInt input1, Int input2, ArrayInt output);
div(ArrayFloat input1, Float input2, ArrayFloat output);
div(ArrayInt input1, ArrayInt input2, ArrayInt output);
div(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters:

1. `input1` is the array to divide.
2. `input2` is the array or value used to divide `input1`.
3. `output` is the result of the division.

Example:

```
ArrayFloat in1(100);
ArrayFloat in2(100);
ArrayFloat res(100);

/* Divide each element of the array by 20 */
div(in1, 20.0, res);

/* Divide in1 by in2 element by element */
div(in1, in2, res);
```

1.5.11 dotProduct

Dot product of two arrays.

```
dotProduct(ArrayInt input1, ArrayInt input2, Float output);
dotProduct(ArrayFloat input1, ArrayFloat input2,
                                              Float output);
```

Parameters:

1. `input1` is the first array.
2. `input2` is the second array.
3. `output` is the result of the dot product, always returned in a float variable.

Example:

```
ArrayInt in1(100);
ArrayInt in2(100);
Float res;

/* Dot product of in1 and in2, the result is put in
   a float variable */
dotProduct(in1, in2, res);
```

1.5.12 energy

Compute the energy, defined as the sum of the squared values of an array.

```
energy(ArrayInt input, Int output);
energy(ArrayFloat input, Float output);
```

Parameters:

1. `input` is the array to process.
2. `output` is the result.

Example:

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute energy of in */
energy(in, res);
```

1.5.13 fft

Compute a Fast Fourier Fourier transform. Parameters of the FFT must be defined in the variable 'fftv'. Input data are overwritten by the results of the transform.

```
fft(FFTInt fftv, ArrayComplexInt array);
fft(FFTFloat fftv, ArrayComplexFloat array);
```

Parameters:

1. `fftv` is the variable containing parameters to process the FFT.
2. `array` is the array to process.

Example:

```
FFTInt fi(128);
ArrayComplexInt ai(128);

/* Compute the fft of the complex array ai */
fft(fi, ai);
```

1.5.14 fir

Finite impulse response filter. The coefficients of the filter must be set in the variable `firv`.

```
fir(FIRInt firv, ArrayInt input, ArrayInt output);
fir(FIRFloat firv, ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `firv` is the variable containing the filter parameters.
2. `input` is the data to process.
3. `output` is the processed data.

Example:

```
FIRFLoat firv(2, 0.1, -0.2);
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute the fft of the complex array ai */
fir(firv, in, res);
```

1.5.15 iir

Infinite impulse response filter. The coefficients of the filter must be set in the variable 'iirv'. The 'array1' must contain the input data, the 'array2' contains the filtered data.

```
iir(IIRInt iirv, ArrayInt array1, ArrayInt array2);
iir(IIRFloat iirv, ArrayFloat array1, ArrayFloat array2);
```

Parameters:

1. `iirv` is the variable containing the filter parameters.
2. `input` is the array to process.
3. `output` is the result of the filter.

Example:

```
IIRInt ii(2, 5,
          1, -2,
          1, -2, 3, -4, 5);
ArrayInt a(100);
ArrayInt b(100);

/* Filter the signal contained in a and put the result in b */
iir(ii, a, b);
```

1.5.16 log10

Common logarithm.

```
log10(Float input, Float output);
log10(ArrayFloat input, ArrayFloat output);
```

Example:

```
ArrayFloat in(5, 1.0, 2.0, 3.0, 4.0, 5.0);
ArrayFloat res(5);

/* Compute the common logarithm of a */
log10(in, res);
```

1.5.17 magnitude

Magnitude of a complex number.

```
magnitude(ArrayComplexInt input, ArrayInt output)
magnitude(ArrayComplexFloat input, ArrayFloat output)
```

Parameters:

1. `input` is the complex array to process.
2. `ouptut` is the magnitude of the input.

Example:

```
ArrayComplexInt in(100);
ArrayInt res(100);

/* Compute the magnitude of the complex number */
magnitude(in, res);
```

1.5.18 max

Find the maximum value and its index in an array.

```
max(ArrayInt array, Int value, Int index);
max(ArrayFloat array, Float value, Int index);
```

Parameters:

1. `array` is the array to process.
2. `value` is the maximum value found in the array.
3. `index` is the index of the array at which the maximum value is found.

Example:

```
ArrayInt in(100);
Int maxVal;
Int iMaxVal;

/* Search the maximum value of the array and put the value
   in maxVal and the index in iMaxVal */
max(in, maxVal, iMaxVal);
```

1.5.19 mean

Compute the mean value of an array.

```
mean(ArrayInt array, Int result);
mean(ArrayFloat array, Float result);
```

Parameters:

1. `array` is the array.
2. `result` is the mean of the array.

Example:

```
ArrayInt a(100);
Int mean;

/* Compute the mean of a */
mean(a, mean);
```

1.5.20 min

Find the minimum value and its index in an array.

```
min(ArrayInt array, Int value, Int index);
min(ArrayFloat array, Float value, Int index);
```

Parameters:

1. `array` is the array to process.
2. `value` is the minimum value found in the array.
3. `index` is the index of the array at which the minimum value is found.

Example:

```
ArrayInt in(100);
Int minVal;
Int iMinVal;

/* Search the minimum value of the array and put the value
   in minVal and the index in iMinVal */
max(in, minVal, iMinVal);
```

1.5.21 mult

Multiply elements of an array with a value or divide two arrays.

```
mult(ArrayInt input1, Int input2, ArrayInt output);
mult(ArrayFloat input1, Float input2, ArrayFloat output);
mult(ArrayInt input1, ArrayInt input2, ArrayInt output);
mult(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters:

1. `input1` is the first operand of the multiplication.

2. `input2` is the second operand of the multiplication.
3. `output` is the result of the multiplication.

Example:

```
ArrayInt in1(100);
ArrayInt in2(100);
Int res;

/* Multiply each element of the in1 array by 534 */
mult(in1, 534, res);

/* Multiply the in1 and in2 array element by element */
mult(in1, in2, res);
```

1.5.22 negate

Negates each value of an array.

```
negate(ArrayInt input, ArrayInt output);
negate(ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the array to negate.
2. `output` is the negative of array.

Example:

```
ArrayInt input(100);
ArrayInt output(100);

/* Negate the input array */
negate(input, output);
```

1.5.23 pow

Power of n.

```
pow(Float input, Float power, Float output);
pow(ArrayFloat input, Float power, ArrayFloat output);
```

Parameters:

1. `input` is the base.
2. `power` is the exponent.
3. `output` is the result.

Example:

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute the power of 12 of each element of the in array */
pow(in, 12, res);
```

1.5.24 rms

Root mean square.

```
rms(ArrayInt input, Int output);
rms(ArrayFloat input, Float output);
```

Parameters:

1. `input` is the array to process.
2. `input` is the root mean square of the array.

Example:

```
ArrayFloat in(100);
Float res;

/* Compute the root mean square of the array */
rms(in, res);
```

1.5.25 sin

Sine.

```
sin(ArrayInt input, ArrayInt output);
sin(ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the array to process.
2. `output` is the sine of each element of the array.

Example:

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute sine of each element of the in array */
sine(in, res);
```

1.5.26 sqrt

Square root.

```
sqrt(ArrayInt input, ArrayInt output);
sqrt(ArrayFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the array to process.
2. `output` is the square root of each element of the array.

Example:

```
ArrayFloat in(100);
ArrayFloat res(100);

/* Compute square root of each element of the in array */
sqrt(in, res);
```

1.5.27 stalta

Short term over long term average. The coefficients of the STA/LTA must be set in the variable `staltav`.

```
stalta(StaLtaInt staltav, Int input, Int output);
stalta(StaLtaFloat staltav, Float input, Float output);
stalta(StaLtaInt staltav, ArrayInt input, ArrayInt output);
stalta(StaLtaFloat staltav, ArrayFloat input,
                                        ArrayFloat output);
```

Parameters:

1. `staltav` is the variable containing the STA/LTA parameters (section 1.3.11).
2. `input` is the input data of the STA/LTA algorithm.
3. `output` is the result of the STA/LTA algorithm.

Example:

```
StaLtaInt sti (400, 4000, 400, 1000);
ArrayInt in(100);
ArrayInt res(100);

/* Compute STLA/LTA from data of the in array */
stalta(sti, in, res);
```

1.5.28 stdDev

Standard deviation.

```
stdDev(ArrayInt input, Int result);
stdDev(ArrayFloat input, Float result);
```

Parameters:

1. `input` is the array to process.
2. `result` is the standard deviation computed from the array.

Example:

```
ArrayInt in(100);
Int res;

/* Compute standard deviation from the in array */
stdDev(in, res);
```

1.5.29 sub

Subtract elements of an array (with a value) or subtract of two arrays.

```
sub(ArrayInt input1, ArrayInt input2, ArrayInt output);
sub(ArrayFloat input1, ArrayFloat input2, ArrayFloat output);
```

Parameters:

1. `input1` is the first operand of the subtraction.
2. `input2` is the second operand of the subtraction.
3. `output` is the result of the subtraction.

Example:

```
ArrayInt in1(100);
ArrayInt in2(100);
ArrayInt res(100);

/* Subtract in2 by in1 element by element */
sub(in1, in2, res);
```

1.5.30 sum

Sum all the elements of an array.

```
sum(ArrayInt array, Int result);
sum(ArrayFloat array, Float result);
```

Parameters:

1. `array` is the array to process.
2. `result` is the sum of each element of the array.

Example:

```
ArrayInt in(100);
Int res;

/* Sum all of the elements of the in array */
sum(in, res);
```

1.5.31 trigger

Return true for a high or low value, or a rising or falling edge, compared to a threshold. The parameters of the trigger must be defined when declaring the variable `triggerv`. The input can be array, in this case the output is set to true if at least one value in the array has produce a trigger.

```
trigger(TriggerInt triggerv, Int input, Bool output);
trigger(TriggerFloat triggerv, Float input, Bool output);
trigger(TriggerInt triggerv, ArrayInt input, Bool output);
trigger(TriggerFloat triggerv, ArrayFloat input, Bool output);
```

Parameters:

1. `triggerv` is the variable containing the trigger parameters (section 1.3.12).
2. `input` is the input data of the trigger algorithm.
3. `output` is the result of the trigger algorithm.

Example:

```
TriggerInt triggerv(RISING EDGE, 2500, 4000, 10000);
ArrayInt in(100);
Bool res;

/* Put true in the res variable if at least one element
   of the in array raised above the threshold defined
   in the triggerv variable */
trigger(triggerv, in, res);
```

1.5.32 var

Variance.

```
var(ArrayInt array, Int result);
var(ArrayFloat array, Float result);
```

Parameters:

1. `array` is the array to process.
2. `result` is the variance of the array.

Example:

```
ArrayInt in(100);
Int res;

/* Compute the variance of the in array */
var(in, res);
```

1.6 Utility functions

1.6.1 ascentRequest

Request the ascent of the float.

```
ascentRequest();
```

No parameters.

Example:

```
ascentRequest();
```

1.6.2 convert

Copy data of a specific type of data to an other type of data.

```
convert(ArrayInt input, ArrayFloat output);
convert(ArrayInt input, ArrayComplexInt output);
convert(ArrayFloat input, ArrayInt output);
convert(ArrayFloat input, ArrayComplexFloat output);
convert(BufferInt input, ArrayInt output);
convert(BufferFloat input, ArrayFloat output);
```

Parameters:

1. `input` is the variable to convert.
2. `output` is the converted variable.

Example:

```
ArrayInt intNumber(100);
ArrayComplexInt complexNb(100);
ArrayFloat floatNumber(100);

/* Convert the integer array into a complex integer array.
   The array to convert is copied into the real part of
   the complex array */
convert(intNumber, complexNb);

/* Convert the integer array into a floating point
   number array */
convert(intNumber, floatNumber);
```

1.6.3 getSampleIndex

This function is only valuable for simulation. It get the index of sample currently read in input data file used for the simulation.

```
getSampleIndex(Int sampeIndex)
```

Parameter:

1. `sampeIndex` is the index of the sample.

Example:

```
Int sampleIndex;
getSampleIndex(sampleIndex);
```

1.6.4 getTimestamp

Get the current date with a resolution of 1 second.

```
getTimestamp(Int timestamp)
```

Parameter:

1. `timestamp` is the current date.

Example:

```
Int timestamp;
getTimestamp(timestamp);
```

1.6.5 record

Record a value, an array, a buffer or a string in a file.

```
record(File file, Int data)
record(File file, Float data)
record(File file, ArrayInt data)
record(File file, ArrayFloat data)
record(File file, BufferInt data)
record(File file, BufferFloat data)
record(File file, String data)
```

Parameter:

1. `file` is the file in which to record the data.
2. `data` is the data to record.

Example:

```
File f;
Int timestamp;
ArrayFloat array;

/* Record the current date and the data in array */
record(f, timestamp);
record(f, array);
```

2 MeLa tutorial

2.1 Continuous recording

The first application continuously records data from the hydrophone. Create a file called `App1.mela` in the `MeLaApps` directory. In the `Compomaid.java` file, write the file name in `String app1_name = "App1";` and leave the second app name blank `String app1_string = "";` (lines 70, 71).

In the `App1.mela` file, define the park ration and the depth of mission, for example:

```
Mission:
    ParkTime: 14400 minutes;
    ParkDepth: 1000 meters;
```

Define the coordinator with an acquisition mode called `ContinuousRecord` executed only during the park stage:

```
Coordinator:
    ParkAcqModes: ContinuousRecord;
```

Define the acquisition mode, the sensor to use, its sampling frequency and the variable that will receive the data from the sensor. The acquisition mode must be a `ContinuousAcqMode` since we want to record all data from the sensor. We use the low frequency hydrophone `HydrophoneBF` with a sampling frequency of 200 Hz and the `x` variable, an array that can contain only one sample.

```
ContinuousAcqMode ContinuousRecord:

Input:
    sensor: HydrophoneBF(200);
    data: x(1);
```

Next, define the file in which to record the data from the hydrophone:

```
Variables:
    File f;
```

A continuous acquisition mode must contain a `RealTimeSequence` that is able to handle all the data from the sensor. In this sequence we call the record function to record the hydrophone data in the file.

```
RealTimeSequence detection:
    record(f, x);
endseq;
```

Finally close the acquisition mode with:

```
endmode;
```

The complete code of the application is:

```
Mission:
   ParkTime: 14400 minutes;
   ParkDepth: 1000 meters;

Coordinator:
   ParkAcqModes: ContinuousRecord;

ContinuousAcqMode ContinuousRecord:

Input:
    sensor: HydrophoneBF(200);
    data: x(1);

Variables:
    File f;

RealTimeSequence detection:
    record(f, x);
endseq;

endmode;
```

Launch the MeLa compiler, it will return the following analysis results:

```
* Verification of execution time:

Maximum processor usage during PARK:
  Error: 10000 % > 100 %
```

```
* Estimation of energy consumption:

Autonomy: 0 years
Energy consumption during DESCENT: 295,3mWh
  Processor consumption: 95,3mWh
  Actuator consumption: 200mWh

Energy consumption during PARK: 16872mWh
  Processor consumption: 2472mWh
  HydrophoneBF: 14400mWh
  Actuator consumption: 0mWh

Energy consumption during ASCENT: 3035,7mWh
  Processor consumption: 35,7mWh
  Actuator consumption: 3000mWh

Energy consumption during SURFACE: 4837510,3mWh
  Processor consumption: 10,3mWh
  Actuator consumption: 0mWh
  Transmission consumption: 4837500mWh


* Amount of data transmitted by satellite:

Transmission per cycle: 675000 kB
Transmission per month: 1915522,56 kB
```

The first thing is that the processor usage is far too high. A way to correct the problem would be to to record the data in a `ProcessingSequence` instead of the `RealTimeSequence` but this would not gives us the guaranty that all the data are recorded conducting to a corrupted signal.

A better solution is to increase the size of the `x` variable, this would reduce the period of execution of the real time sequence, giving more time to the recording function to be executed. Indeed the recording function has an execution time almost independent of the amount of data to record.

Define the size of the `x` variable to 10000. The processor usage is now only of 1%. However the power consumption is still high. We can see that most of the power is consumed at surface because all the recorded data are transmitted by satellite. Instead of recording all the data it is possible to record only some detected signal. Thus, our next objective is to write a simple detection algorithm.

2.2 Detection algorithm

The principle of this detection algorithm is to detect a raise of amplitude in the acoustic signal and to record 100 seconds before and 200 seconds after this raise.

The sampling frequency of the hydrophone is set to 20 Hz for this application, the lowest is the sampling frequency, the lowest will be the amount of recorded data. We will process the data by packets of 1 seconds, which means that we will be able to detect the beginning of a seismic wave every second. Choosing a smaller packet size could increase the processor usage (but not necessary) and big blocks of data would not allow us to identify the beginning of a seismic wave. The input part of the acquisition mode must be now:

```
Input:
    sensor: HydrophoneBF(20);
    data: x(20);
```

In order to record 100 seconds before and 200 seconds after the amplitude raise, we need a buffer that will keep 300 seconds of data in memory. For a sampling frequency of 20 Hz, the buffer have to be able to contain 6000 samples. Each data packet have to be appended to the buffer with the push function. The content of the acquisition mode must be now:

```
Input:
    sensor: HydrophoneBF(20);
    data: x(20);

Variables:
    /* Buffer containing 300 seconds of data,
       or 6000 samples at 20 Hz */
    BufferInt last5Minutes(6000);

RealTimeSequence detection:
    push(last5Minutes, x);
endseq;
```

Now we use an STA/LTA algorithm to detect a raise of amplitude in the signal. The result of the STA/LTA gives a value close to 1 when the signal is stable but this value increase when there is an elevation of the acoustic signal that reaches the Short Term Average. Considering that the duration of an earthquake is of several hundred of seconds, we want to compute the sort term average over 10 seconds of data, which is 200 samples. For the long term average we choose a length of 100 seconds that does not include the 10 seconds of the short term average. These parameters will be refined later. Since the average is computed with integers we have to define a scaling factor, we choose it to be 10 which is sufficient to see changes of the ratio at a resolution of 0.1. The content of the acquisition mode must be now:

```
Variables:
    File f;
    /* Buffer containing 300 seconds of data,
       or 6000 samples at 20 Hz */
    BufferInt last5Minutes(6000);

    /* STALTA variables */
```

```
    StaLtaInt staltaInstance(200, 2000, 200, 1000);
    ArrayInt staltaResult(200);

RealTimeSequence detection:
    push(last5Minutes, x);
    stalta(staltaInstance, x, staltaResult);
endseq;
```

The result of the STA/LTA is used to trigger the recording of the signal, however it is not sufficient to compare the STA/LTA result with a threshold value since it would record the signal several times while the threshold is exceeded. The signal must trigger the recording only one time. Thus we define a `TriggerInt` variable called `triggerInstance` with the `RISING_EDGE` trigger mode. We want to activate the trigger on a rising edge with a threshold of `2000` (equivalent to 2 when we consider the scaling factor of 1000 of the STA/LTA). Since we want to wait 150 seconds after the trigger to get the whole seismic signal, we define the trigger delay to 3000 samples. We also define a minimum time between each trigger such that two records must be spaced by at least 60 seconds (1200 samples).

```
/* Trigger variable */
TriggerInt triggerInstance(RISING_EDGE, 2000, 3000, 1200);
```

The `trigger` function must be put in the real time sequence after the `stalta` function. Its parameters must include the `triggerInstance` variable, the `staltaResult` variable and a boolean variable that will be set to true each time the trigger is activate. The recording function is put in a condition that test the boolean variable. The condition must be annotated with a probability, in this case `1 per day`, that is used by the application to compute the amount of data that will be transmitted trough satellite. The recording function can not be put in the real time sequence because its execution time is too long. Instead, we put it in a processing sequence for which functions with long execution time are allowed (at the cost of potentially missing some samples). The final application is described bellow:

```
Mission:
   ParkTime: 14400 minutes;
   ParkDepth: 1000 meters;


Coordinator:
   ParkAcqModes: DetectionRecord;


ContinuousAcqMode DetectionRecord:

Input:
    sensor: HydrophoneBF(20);
    data: x(20);
```

```
Variables:
    File f;
    /* Buffer containing 300 seconds of data,
       or 6000 samples at 20 Hz */
    BufferInt last5Minutes(6000);

    /* STALTA variables */
    StaLtaInt staltaInstance(200, 2000, 200, 1000);
    ArrayInt staltaResult(200);

    /* Trigger variable */
    TriggerInt triggerInstance(RISING_EDGE, 2000, 3000, 1200);
    Bool trigRes;

RealTimeSequence detection:
    push(last5Minutes, x);
    stalta(staltaInstance, x, staltaResult);
    trigger(triggerInstance, staltaResult, trigRes);
    if trigRes:
        @probability = 1 per day
        call recordSeq
    endif;
endseq;

ProcessingSequence recordSeq:
    record(f, last5Minutes);
endseq;
endmode;
```

We do not show the simulation part allowing to set the parameters of the application.

2.3 Short acquisition

Instead of using a detection algorithm, one may want to record only some small snippets of the signal. For this we can use a short acquisition mode. The time interval at which the acquisition mode is executed must be set in the coordinator. Each time we record 10 minutes of data (200 samples).

```
Mission:
   ParkTime: 14400 minutes;
   ParkDepth: 1000 meters;

Coordinator:
   DescentAcqModes: ShortRecord every 60 minutes;
   ParkAcqModes: ShortRecord every 1 hour;
   AscentAcqModes: ShortRecord every 60 minutes;
```

```
ShortAcqMode ShortRecord:

Input:
    sensor: HydrophoneBF(200);
    data: x(600);

Variables:
    File f;

ProcessingSequence recordSeq:
    record(f, x);
endseq;

endmode;
```

2.4 Composition

In order to execute both application on the same instrument we can use the composition tool that will take two MeLa files or just copy both applications in the same file and edit the coordinator part to start the acquisition modes at the appropriate times. The corresponding MeLa application will be:

```
Mission:
   ParkTime: 14400 minutes;
   ParkDepth: 1000 meters;

Coordinator:
   DescentAcqModes: ShortRecord every 60 minutes;
   ParkAcqModes: ContinuousRecord
                 ShortRecord every 1 hour;
   AscentAcqModes: ShortRecord every 60 minutes;

ShortAcqMode ShortRecord:
    ...
endmode;

ContinuousAcqMode ContinuousRecord:
    ...
endmode;
```