

Contents

Welcome

Training days

Latest software

Here to help

Script Spotlight

- Sequencer library



- Changing or hiding markers
- Script – Independent DAC control

Signal

- Dynamic clamp available models
- Script – Info window from data view

Scripters corner

- Flow of control statements

Recent questions

- Labelling data

CED user forums

Welcome

Thank you for downloading our December newsletter. As 2020 draws to a close we look back and reflect on all the new challenges the year has brought. Despite the problems caused by the pandemic, we have managed to both stay safe and keep working to provide you with the software and hardware to continue your research. We are fortunate that no-one here has been seriously ill, and we hope this is the case for you and yours. We have continued providing as much of our customary support as possible without physically visiting your labs and the Software team have kept busy. Spike2 Version 10.08 was released recently (so make sure to update your copy from our [website](#)). Signal 7.06 is coming soon.

We are also excited to announce a new Talker for the BrainVision actiChamp Plus amplifier from Brain Products. This talker bridges Spike2 to the amplifier, allowing you to capture data with Spike2 from your existing amplifier without the CED1401. The updated version 2.0 of our Delsys Talker has been released, and we have new Talkers for additional equipment on the horizon. Watch this space!

As always, should you have any suggestions, questions or queries for the newsletter please get in touch Marjorie@ced.co.uk.

From all of us at CED, we wish you Happy Holidays and a Happy (and Healthy) New Year.

Training Days

Due to the ongoing COVID-19 pandemic, all current training events have been put on hold. We do however offer remote training sessions by Skype/Zoom/Teams either one-to-one or with groups.

Join us and learn how to make the best use of Spike2 and Signal to save hours of repetitive analysis. Our remote sessions are free to arrange and are suitable for both existing and prospective users of our data acquisition and analysis systems. If you would like to schedule a session, please get in touch.

If you are interested in hosting a training event in your local area once social distancing measures have been eased, please get in touch: Marjorie@ced.co.uk.

If you see these buttons in our newsletters, it means a file or script relating to the section is available to download:



Latest versions of Spike2 and Signal

Spike2	Released	Signal	Released
Version 10.08	11/2020	Version 7.05a	02/2020
Version 9.12	09/2020	Version 6.05b	10/2019
Version 8.20	09/2020	Version 5.12a	02/2018
Demo	11/2020	Demo	02/2020

[Back to contents](#)

Here to help

We know access to your labs has been erratic for the past 8 months, and with the local lockdown measures that is unlikely to change over winter. However, CED will still do all in our power to support you for increased home working. Should you require any help or wish to discuss your system, email us at info@ced.co.uk and we can arrange for a video call via Skype/Zoom/Teams.

We also have tutorial videos for both Spike2 and Signal available on our [website](#) for you to peruse at your leisure. There are new videos and updates in the pipeline, however if you have a particular topic you think could benefit from a tutorial video please let us know.

[Back to contents](#)

Script Spotlight – Sequencer library / Seqlib.s2s

The output sequencer of Spike2 and Signal is used to generate digital and waveform outputs from your 1401 during sampling. This tool can be used to trigger and control a wide range of devices such as electrical and magnetic stimulators, motors, or visual stimulus generators. Often you need to be able to set these stimulus parameters whilst recording and, while we have made it possible to load in replacement sequencer files during sampling, sometimes it is more convenient to use a script that allows the user to input various stimulus parameters with a dialog. The script then passes these new values to the sequencer to update the experiment protocol in real time.

Simple as it sounds, many script writers come unstuck when it comes to re-calculating user entered values as sequencer values. The sequencer runs in real time and uses a rather 'low-level' language to control digital input and output and the DACs. When you want to set these values with sequencer variables the actual values to set are not always obvious. For example, the command to set a DAC to 3.0 Volts involves converting the desired user-units output to a 32-bit integer value. The full 32-bit range of the sequencer variable value corresponds to the full range of voltages that can be generated by the DAC, so -2147483648 corresponds to the lowest possible output voltage (-5 or -10 volts), 0 corresponds to a 0 output and 2147483647 corresponds to the highest possible output voltage. Therefore, 3V is equal to $(2147483647 / \text{ADC range}) * 3$.

We have provided a script to make these calculations as stress-free as possible. Seqlib.s2s is stored in the *include* folder of your Spike2 and Signal installation and contains a library of functions to help with the calculation of sequencer variables. Included is a function to convert user entered units into 32-bit numbers for use by the DAC output commands. This script is intended for use as an include file so you can simply enter `#include <seqlib.s2s>` at the top of your own script to add it.

[Back to contents](#)



How can I change or hide a particular marker in my data?

It is possible to change the marker code in Spike2 or filter any channel that holds marker, WaveMark, TextMark or RealMark data through two dialogs.

Set Marker codes for items in a channel

To change existing markers to a new code we use the Set marker codes dialog. This is accessed from the *Analysis* menu or by right clicking on a marker channel. Each data item has four marker codes with 256 values that can be set. Typically, using only the first marker code is enough for most applications.

The dialog box is titled "Set marker codes for Demo.smr[32-bit]". It contains the following fields and controls:

- Channel:** A dropdown menu showing "4 Textmark (TextMark)".
- Start time:** A text field with "0.0".
- End time:** A dropdown menu showing "MaxTime()".
- Status:** A text area showing "Set changes code 0=00 in 16 items on channel 4. Time range 0 to 120.921 seconds."
- Set these codes:** Four checkboxes labeled 0, 1, 2, and 3. Checkboxes 0 and 1 are checked.
- To these values:** Four spinners showing "00" for each code.
- Buttons:** "Help", "Close", and "Set".

- The *Channel* field sets the channel to process.
- *Start time* and *End time* identify the range for markers to set.
- The number of markers within range are noted underneath the time fields.
- The four check boxes 0 to 3 select the codes to change.
- Set the value for each code as either two hexadecimal digits or one printing character.
- If a marker filter is set for the channel, only data items that are in the filter are changed by this dialog.

Marker filter

Each marker channel has a marker filter that selects the data items to display and use in calculations. To access the marker filter, right click a marker channel and select *Marker Filter* or alternatively use the *Analysis* menu > *Marker Filter* option. If you set a marker filter so that it would not pass all data, the channel number displays in red.

The dialog box is titled "Demo.smr[32-bit] marker filter". It contains the following fields and controls:

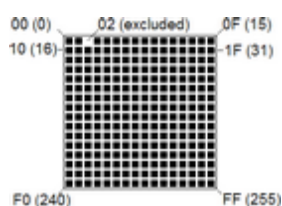
- Channel:** A dropdown menu showing "4 Textmark (TextMark)".
- Mode:** A dropdown menu showing "All masks, all codes must match".
- List format:** A dropdown menu showing "Printing character or hexadecimal".
- Waveform reads use trace number:** A checkbox that is checked.
- Filter specification as text:** A text field.
- Visual representation:** A grid showing four marker codes (0, 1, 2, 3) for each data item. Codes 0 and 1 are checked.
- Buttons:** "All", "None", "Invert", "Copy", "Paste", "Update", "Help", "Cancel", and "OK".

- Each marker data item has four marker codes that are matched against the marker filter.
- You can specify a marker filter using text or by selecting the current layer mask and checking the codes in the check box list.
- *Copy* the current dialog filter to the clipboard, as text.
- *Paste* will set the dialog filter state to match the text filter held on the clipboard.
- *Update* the channel display to correspond with the new filter.
- *All* will enable all markers for no set filter.
- *None* filters all markers, for none to be shown.
- *Invert* the current filter selection (all filtered are now enabled and vice versa).

Text version of filter

The dialog displays the text version of the filter above the 4 filter masks. You can edit the text version and when it is legal, the filter is applied to the masks and the check box list. If you make a change to the check box list, the text updates to match. If you edit the text to an illegal state, an error message appears at the bottom of the dialog.

Masks



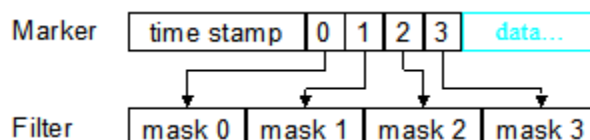
The dialog shows the marker codes as four masks numbered 0 to 3 which give a quick indication of the state of the filter. Each mask has 256 elements in a 16x16 grid, one for each possible code value. The contents of the front-most mask are displayed in the scrolling list in the centre of the dialog. Click on a mask element to bring the mask to the front and scroll the list to the element. If you edit the text version of the filter, the first mask changed by the edit comes to the front of the stack.

The top row of each mask represents code values 0 to 15 (hexadecimal codes 00 to 0F), the second row 16 to 31 (10 to 1F) and so on down to the bottom row, which represents values 240 to 255 (F0 to FF). If a code value is included in the filter, the corresponding element is black. When values are excluded, the element is white.

Modes

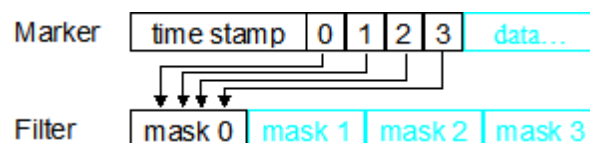
Mode 0 (AND): All masks, all codes must match

You can think of this as the AND mode; to accept data marker code 0 must be in the mask 0 AND marker code 1 must be in mask 1 AND marker code 2 must be in mask 2 AND marker code 3 must be in mask 3. Most users of this mode set mask layers 1, 2 and 3 to All and use the first layer to select data values.



Mode 1 (OR): One mask, any code can match

This mode can be considered as the OR mode; to accept data marker code 0 OR marker code 1 OR marker code 2 OR marker code 3 must be in mask 0. There is one exception; for marker codes 2 to 4, the code 00 is ignored. To accept code 00, it must be the first marker code.



Mode 1 is often used when sorting spike shapes (WaveMark data) and you discover a WaveMark that is the result of a collision between two spikes. You can set the first marker code to the code for the first spike and the second to the code for the second (leaving the third and fourth codes as 00), then the spike will appear on screen and in analyses when either of the codes are included in the mask. We will be discussing the marker filter with WaveMarks in more detail in a further newsletter.

[Back to contents](#)

Scripts: Spike2

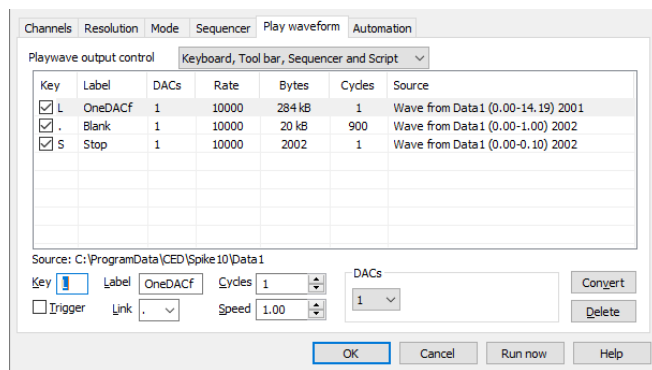
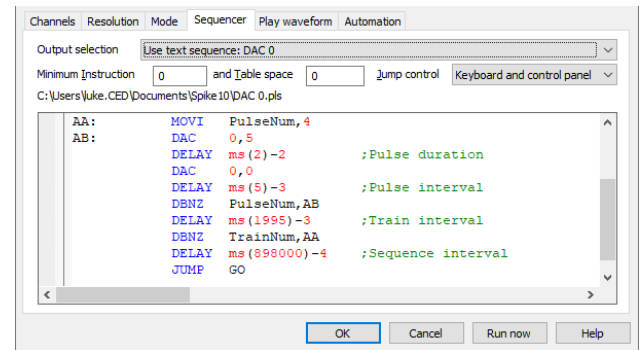
The script [Independent DAC control.s2s](#) was written for a user to solve the problem of presenting two independent pulse trains with different periods. When creating pulse trains within the built-in sequencer, the trains are synchronised across all analogue outputs. If you needed to have two DAC outputs for your experiment, typically you would design three sections in the graphical sequencer: the first with DAC 0 active, the second with DAC 1 active, and the third with both DACs active. These may be linked to sample keys to swap between the sections as needed.

This user needed complete independent control over two pulse trains, with the control of one not affecting the output of the other. There is therefore no fixed interval between the two outputs, nor are the start times of the trains. Whilst this is possible to achieve with the 1401 digital outputs (see the DIGPS, DIGPC, and DIGPBR instructions of your Spike2 online help added in version 10.06), the pulse trains would be driving independent stimulators and

needed the option of either monophasic or biphasic pulses. Each DAC output would consist of several small trains of pulses, repeated at a few seconds interval for a set number of times. There would then be a large delay of several minutes whereupon the train is repeated, and so on until the end of the experiment.

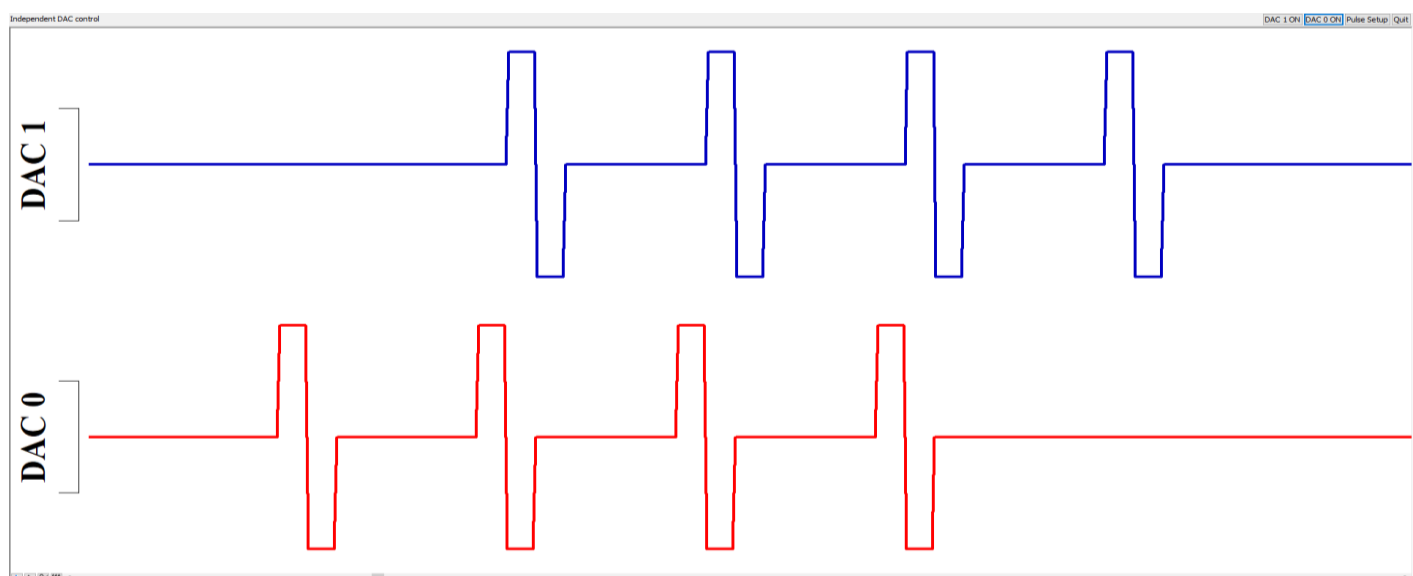
To achieve this, our script writers made use of both the text sequencer output and the Playwave output. The sequencer output generates waveform output from the 1401 DACs either by using the graphical or text version. By using the text sequencer, we were able to store all instructions within the script and create a new sequencer file based upon the user's parameters. The Playwave output operates differently. Instead of using instructions to generate waveforms, it stores waveforms in memory when sampling to playback upon request. By using both the sequencer and Playwave outputs we mitigate the issue of instruction overlap. With the waveform for the Playwave output stored in the 1401s memory during sampling, and the sequencer generating a waveform through instructions, they are both able to play out of individual DAC outputs at the same time.

The script operates by first generating a dialog for the user to input their pulse train parameters. The `SequencerCreate%` function then generates a new text sequencer file and prints the sequencer instructions with the variables updated from the user's parameters. This sequencer file is saved and set as the current instructions for playing out of DAC 0.



The `PlayWaveMake%` function creates a virtual channel holding the defined train from the user's parameters. This is added to the Playwave function as well as a short section of no output. The train and zero output are linked to one another within the Playwave function, with the number of repeats of the zero-output adding up to the defined delay by the user. By repeating small sections of waveform we keep the amount of memory used by the Playwave function to a minimum. The Playwave is set to play out of DAC 1.

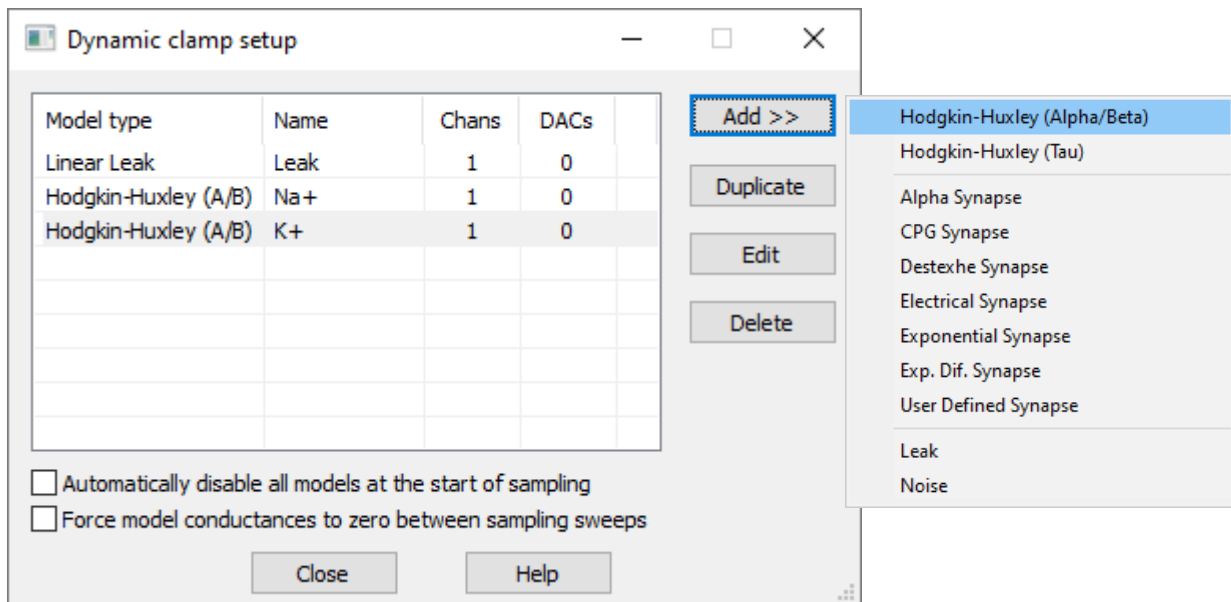
The idle function of the script keeps track of the status of both the sample sequencer and Playwave output. The pulse trains for both DAC 0 and DAC 1 are linked to *Sample Keys* and may be controlled by pressing the correct key on the keyboard. However, we have linked these to their own toolbar buttons which report the status of the output (ON/OFF) automatically. The `DAC0%` and `DAC1%` functions perform the opposite action of the current status (i.e. if OFF then turn ON) by sending the corresponding sample key virtually with the `SampleKey()` script command.



Signal

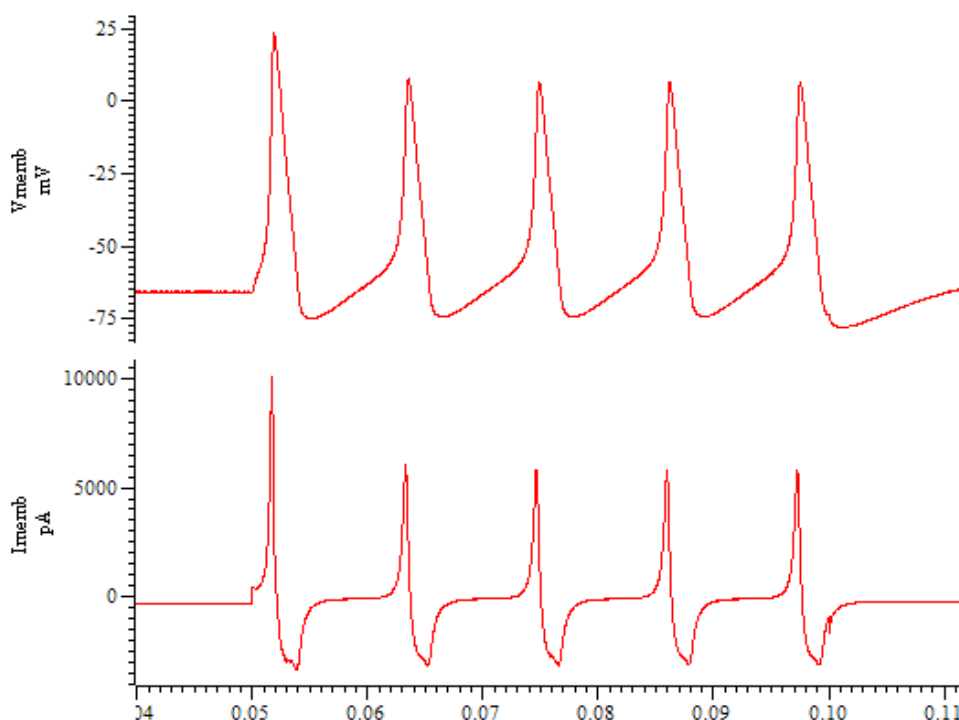
I am interested to know which models are supported by the dynamic clamp system in Signal.

The Dynamic Clamp system in Signal is configured from the *Clamp* tab of the sampling configuration dialog. The clamp section is enabled or disabled from the *Clamp* tab of the *Edit* menu > *Edit Preferences* dialog and provides the main setup dialog which is used to define the models to use during an experiment. This dialog can be accessed on-line to enable or disable models and to view, edit and update model parameters while sampling. Up to 15 models of any type can be defined for use on-line in a single sampling configuration. The number of models that can run simultaneously during sampling is dependent on the sample rate and model complexity.



Adding models to the Dynamic clamp setup

Below is a data file showing action potentials simulated using the combination of models in the setup dialog above.



Simulated action potentials



Hodgkin-Huxley

These models are used to simulate the membrane conductance generated by a population of voltage-dependent ion channels that obey the standard Hodgkin-Huxley equations:

- *Hodgkin-Huxley (Alpha/Beta)* uses a formulation based upon multiple rate equations.
- *Hodgkin-Huxley (Tau)* uses a formulation based on thresholds, threshold sensitivities and time constants.

Synapse

These are used to simulate synaptic current flow between two cells:

- *Alpha synapse* uses the Alpha function to generate the post-synaptic conductance profile.
- *CPG synapse* mimics the behaviour of synapses in the central pattern generation region of the lobster's central nervous system. The pre-synaptic potential mediates the release of a transmitter which causes a post-synaptic current to flow.
- *Destexhe synapse* uses the Destexhe model to generate the post-synaptic conductance profile. The synapse can be activated by the pre-synaptic potential being above or below a defined threshold or by an external TTL pulse or internal timing.
- *Electrical synapse* simulates a simple electrical connection (a gap junction) between two cells.
- *Exponential synapse* simulates the presence of a synapse using an instantaneous rise time and a single exponential to mimic the post-trigger decay of the conductance.
- *Exponential difference synapse* generates the post-synaptic conductance profile based on the difference between two exponentials, one used to model the rising phase and the other to model the decay.
- *User-defined synapse* generates a post-synaptic conductance based on a user-defined waveform read from a text file.

Leak models

These provide simpler dynamic clamp behaviours where the simulated conductance is not time-dependent:

- *Linear leak* represents current leakage through a cell membrane with constant conductance.
- *Goldman-Hodgkin-Katz (GHK) leak* simulates the behaviour caused by diffusion of an ion through a membrane via always-open ion channels, driven by the differing ionic concentrations either side of the membrane and the voltage difference across it.
- *Boltzmann leak* resembles a linear leak with an additional factor in the equation that represents a voltage-dependent blockage of the leak.

Noise

This model can be used to study the effect of simulated noisy conductance on neuronal behaviour:

- *Ornstein-Uhlenbeck noise* generates a noise model using the mean reverting stochastic process.

Full details of the available models, and the mathematics used to generate them, are available in the Signal on-line help.

Scripts: Signal



We have developed a useful script to display waveform information in an info window whilst sampling. Info windows display text, times and other information using a large font for extra visibility across the lab. The individual windows are attached to the current sampling view, where it takes its information from.

Currently the [Infofromdata.sgs](#) script reports the two most recent peak values according to user defined thresholds. The script operates by using an active cursor to search for key features in the data, in this case peaks, and reporting the value at the cursors position using the ChanValue() script command. The info window showing the value can be stretched or moved as desired.

This script could easily be edited to display other information such as rising/falling thresholds, troughs, peak-to-peak value etc. This is achieved with two cursors and the ChanMeasure() script command to report similar information gathered by the Cursor Regions dialog. If your system supports text to speech, it is also possible to configure the script to convert the reported string into speech using the Speak() script command.

We are interested in hearing your comments about the type of information you would like reported during sampling, or of any other functions you would like to scripts produced for; please email us at Marjorie@ced.co.uk.

[Back to contents](#)

Scripters Corner – Flow of control statements

In our previous newsletter we covered arrays of simple variables and touched on the `for... next` looping statement. How the script chooses the next statement to execute is known as the 'flow of control', and these statements let scripts loop and branch. There are two branching statements, `if...endif` and `docase...endcase`, and three looping statements, `repeat...until`, `while...wend` and `for...next`. The following is an example of a branch or 'conditional execution', that directs the flow of control using the 'if' statement:

```
'Example: if
var num%;
num% := Input("Type in an integer number please", 0);
if num% < 0 then Message("It was negative!"); endif;
if (num% mod 2) = 1 then
    Message("It was odd!");
else
    Message("It was even!");
endif;
Halt;
```

The `if` statement is used in two ways; when used without an `else` a single section of code is executed conditionally. When used with an `else`, one of two sections of code is executed. It is considered good practice to keep flow of control statements on separate lines, but the script syntax does not require this as demonstrated with the first `if` statement of the above example. The `Input()` function obtains an integer from the user and stores it in the variable `num%`. The `if` statement directs the flow of control to the required place depending on whether the expression evaluates to 'true' or 'false'. The expression `(num% mod 2)` is the remainder when `num%` is divided by 2.

If you need more than two alternative branches, the `docase` statement is usually more compact than nesting many `if` statements. The `docase...endcase` keywords enclose a list of case statements forming a multi-way branch:

```
'case.s2s
var i%,m$;
i% := ViewKind();      'get type of the current view
docase
  case i% = 0 then m$ := "time";
  case i% = 1 then m$ := "text";
  case i% = 2 then m$ := "output sequence";
  case i% = 3 then m$ := "script";
  case i% = 4 then m$ := "result";
  case i% = 8 then m$ := "external text";
  case i% = 9 then m$ := "external binary";
  case i% = 10 then m$ := "Spike2/Signal";
  case i% = 12 then m$ := "XY view";
  else m$ := "something else...";
endcase;
message("Current window is of type "+m$);
```

The above example displays the type of a file handle. Each `case` statement is scanned until a match (non-zero) is found, or if no match is found the flow passes to the `else` statement. If the `else` is omitted, control passes to the statement after the `endcase` if no `case` expression is non-zero. Only the first non-zero `case` is executed (or the `else` if no case is non-zero). The `ViewKind()` script function obtains the type of the current view as an integer and passes it to the variable `i%`. The returned integers 0 to 12 indicate the type of view as seen in the above example.

Looping statements direct the flow of control by performing a sequence of statements and looping back to the beginning once the end is reached. The three looping statements each control the flow in a unique way. First up is `repeat ... until`:

```
'Example: Mean1
var n, mean, total;
var count% := 0;
repeat
  n := Input("Please input a value", 0.0);
  if n <> -999 then
    total := total + n;
    count% := count% + 1;
  endif;
until n = -999;
if count% > 0 then
  mean := total / count%;
  PrintLog("Mean is %f\n", mean);
else
  PrintLog("No numbers entered...\n");
endif;
Halt;
```

When this example script runs, it prompts you to enter real numbers until you enter -999. On detecting -999 the script calculates the mean of the numbers. The `total` variable holds the sum of the entered numbers; `count%` holds how many numbers have been entered. Dividing `total` by `count%` forms the mean.

The `PrintLog()` script function has been used in our examples before, however in the above you will notice the extra 'flag' `%f`. The `%f` means 'print the value of a real variable here'. It is known as a format specifier: other format specifiers begin with `%` and include `%d` ('print the value of an integer variable here') and `%s` ('print the value of a string variable here'). The variables to print are listed as further arguments to the `PrintLog()` function. In the above example, 'mean' is the variable to be printed. The `\n` after printing the mean in the `PrintLog()` statement is a new-line character. A similar printing code is `\t`, which tells the script to print a tab character.

Next, we have a similar example using `while...wend`:

```
'Example: Mean2
var n, mean, total;
var count% := 0;
n := 0.0;
while n <> -999 do
    n := Input("Please input a value", 0.0);
    if n <> -999 then
        total := total + n;
        count% := count% + 1;
    endif;
wend;
if count% > 0 then
    mean := total / count%;
    PrintLog("Mean is %f\n", mean);
endif;
Halt;
```

Note that the value of `n` must be set to 0.0 initially to get into the loop. If `n` were initially set to -999 the loop would never be executed. In contrast the `repeat...until` example the loop is always executed at least once.

Finally, an example using `for...next`:

```
'Example: Mean3
var n, mean, total;
var count%;
for count% := 1 to 4 do
    n := Input("Please input a value", 0.0);
    total := total + n;
next;
mean := total / 4;
PrintLog("Mean is %f\n", mean);
Halt;
```

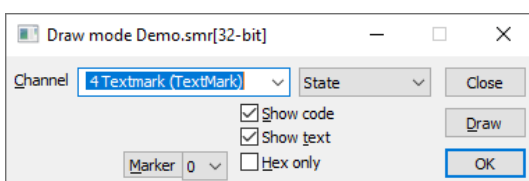
The `for...next` statement simply controls the flow by stating for each of the integers from start to end the script will perform the code between `for` and `next`. This time we loop around the `Input()` statement four times as `count%` takes a value from 1 to 4.

[Back to contents](#)

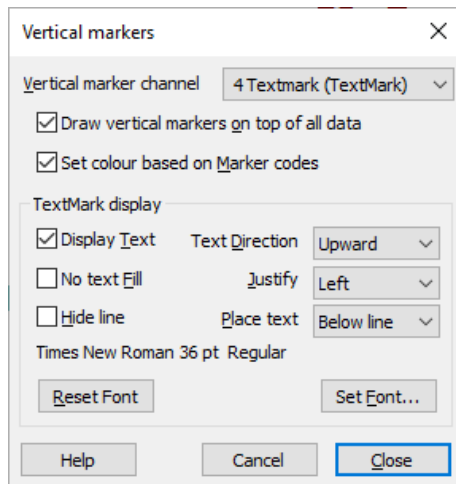
Recent Questions – *Is it possible to add visible labels to my data after recording?*


By using TextMarks we can either use the State drawing mode or the Vertical markers function to act as labels. TextMarks hold marker information alongside user-defined text at a specific time stamp, and this text may be displayed on screen (below).

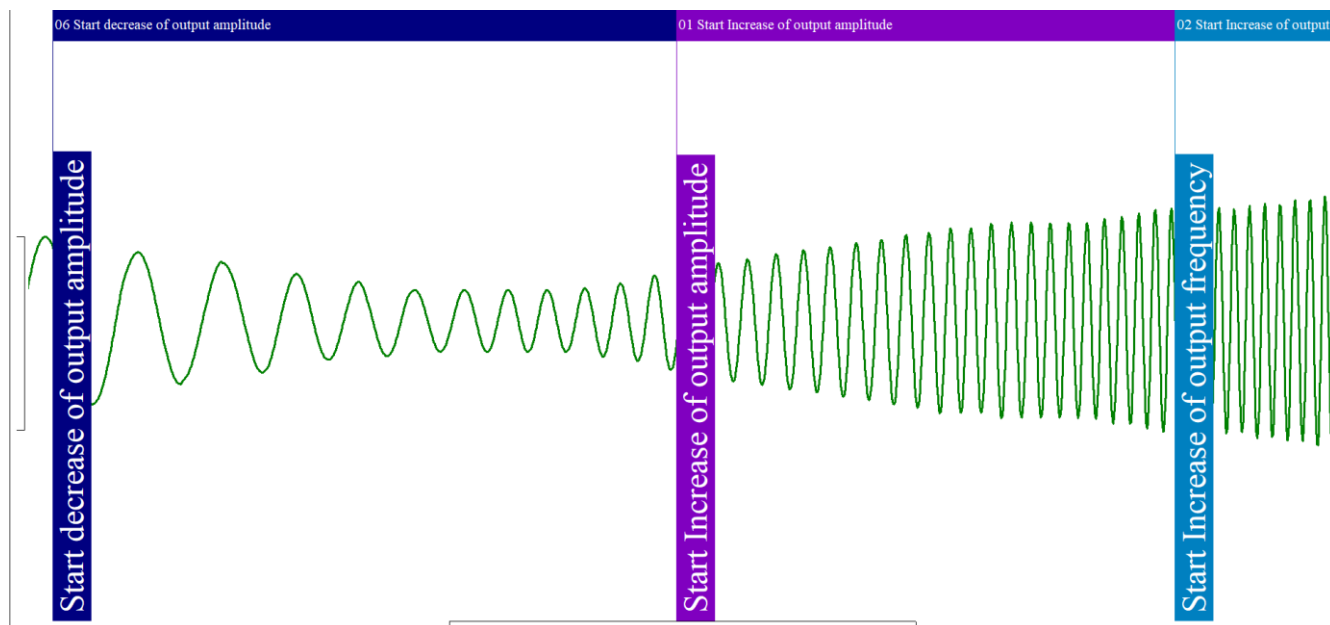
To create a TextMark channel specifically for labelling after sampling, add a new memory channel by going to the *Analysis* menu > *Memory Buffer* > *Create new buffer* and selecting *TextMark* as the buffer type. Use the *Import* function to create TextMarks from existing Events or press *OK* to create a new blank buffer. The *Analysis* menu > *Memory Buffer* > *Add items* is used to add new TextMarks. Note that memory buffers are stored in memory and will be lost on closing the data file unless saved to disk with the *Analysis* menu > *Memory Buffer* > *Write to disk* dialog.



Alter the drawing mode to State by right clicking the channel and selecting *Drawing mode*. Choose the *State* option from the drop-down menu and *Show text* with the tick box. *Show code* will display the associated marker code alongside the text.



Access the *Vertical markers* dialog with the *View* menu > *Vertical markers* option. From this dialog select the *Draw vertical markers* option and then *Display Text*. Adjust the *Text Direction*, placement, and set the font as desired. If *Set colour...* is enabled, the colour of the vertical markers is dependent on the marker code originally chosen. The marker code may be altered with the *Analysis* menu > *Set Marker codes*; alternatively, the colour for all markers using the code may be changed with the colour palette accessed from the toolbar. 



[Back to contents](#)

CED User forums

Try the [CED Forums](#) bulletin board for software and hardware support.

If you have any comments about the newsletter format and content, please get in touch: Marjorie@ced.co.uk.

To adjust your subscription preferences, please visit our website: www.ced.co.uk/upgrades/subscribeenews.

[Back to contents](#)

Contact us:

In the UK:

Technical Centre, 139 Cambridge Road,
Milton, Cambridge, CB24 6AZ, UK
Telephone: (01223) 420186
Fax: (01223) 420488

Email:

info@ced.co.uk
International Tel: [44] 1223 420186
International Fax: [44] 1223 420488
USA and Canada Toll Free: 1 800 345 7794
Website: www.ced.co.uk

All Trademarks are acknowledged to be the Trademarks of the registered holders.
Copyright © 2020 Cambridge Electronic Design Ltd, All rights reserved.