Spike2
Technical Notes

July 2009

Copyright © Cambridge Electronic Design Limited 2001, 2009

Neither the whole nor any part of the information contained in, or the product described in, this guide may be adapted or reproduced in any material form except with the prior written approval of Cambridge Electronic Design Limited.

Version 1	September 2001
1401-18 added	July 2009

Published by:

Cambridge Electronic Design Limited
Science Park
Milton Road
Cambridge
CB4 0FE
UK

Telephone:	Cambridge (01223) 420186
Fax:	Cambridge (01223) 420488
Web:	http://www.ced.co.uk
Email:	info@ced.co.uk

Trademarks and Trade Names used in this guide are acknowledged to be the Trademarks and Trade Names of their respective Companies and Corporations.

Table of contents

The ArrFFT command

Several users have been confused by our use of the FFT in the `ArrFFT()` command in both Spike2 and Signal. The confusion arises because of two factors:

1. We transform between a purely real time series and a complex frequency series
2. We show results only for positive frequencies in the forward and inverse transforms so that the result of a transform occupies the same space as the original data.

This short document attempts to explain what we actually do, and how to obtain results for positive and negative frequencies if you require it. Results are quoted, not proved. If you need more information about the mathematics of this subject, see the references at the end.

Discrete Fourier transforms

The inverse discrete complex Fourier transform (inverse DFT) is a one to one mapping of any series of complex numbers $A(n)$, $n=0,1,2,\dots N-1$ onto another series defined by:

$$X(j) = \sum_{n=0}^{N-1} A(n)\, W_N^{nj},\quad j=0,1,2,\dots N-1$$

where $W_N = \exp(2\pi \boldsymbol{i}/N)$ and $\boldsymbol{i}$ is the square root of -1. The relationship that gives the $A(n)$ in terms of the $X(j)$ is the familiar discrete Fourier transform (DFT):

$$A(n) = \frac{1}{N} \sum_{j=0}^{N-1} X(j)\, W_N^{-nj},\quad n=0,1,2,\dots N-1$$

It is a matter of convention where the factor of $1/N$ goes. As shown above, the A(n) are the amplitudes of the constituent sine and cosine waves. Some implementations place the $1/N$ in front of the inverse transform.

Circular nature of the transform

An important property of this pair of transforms is that the indices n and j are to be interpreted modulo N. That is

$$X(j + kN) \equiv X(j)$$

for all integral values of k (positive or negative). W_N^{nj} expands to $\exp(\boldsymbol{i}2\pi nj/N)$ which is equivalent to $\cos(2\pi nj/N) + \boldsymbol{i}\sin(2\pi nj/N)$. Adding any multiple of N to j or k clearly makes no difference to the value.

Fast Fourier transforms

You will notice that the above discussion makes no statements about the value of N. In fact, it is true for any positive N. However, the calculation of the transform as written is a process of order N^2, which reduces its practical usefulness. However, if the number N is highly composite (has many divisors), the process can be greatly speeded up. For the case of N a power of two, the process becomes of order $N \log_2(N)$.

The Fast Fourier transform, or FFT, is simply the calculation of a discrete Fourier transform using a highly composite N. Most implementations of it (including ours) make N a power of two, but it need not be as long as N is the product of reasonably small numbers compared to the original N. All the following assumes that N is a power of 2.

Negative frequencies

The following discussion assumes that we are transforming data between the time domain $X(j)$ and the frequency domain $A(n)$. The circular nature of the time and frequency indices n and j discussed above allow us to consider the frequency domain to run from either 0 to N-1 or from -N/2 to N/2, the latter being the more usual. You should notice that both the 0 and N/2 frequency components are shared between the positive and negative frequencies (which is why there are only N components in a frequency range from -N/2 to N/2, and not N+1).

In terms of sampled data, the frequency domain runs from minus half the sampling rate of the data to plus half the sampling rate. It seems strange that the DFT of a sinusoid produces the surprising result that it consists of one sinusoid at a positive frequency and one at a negative, each of half the amplitude you might expect. In fact, all the maths is saying is that both answers fit the data and it cannot choose between them.

Symmetry relationships

If the time series $X(j)$ is wholly real, the $A(n)$ must be conjugate even. Likewise, if $X(j)$ is wholly imaginary, the $A(n)$ must be conjugate odd. That is, if we use * to represent the complex conjugate, if $X(j)$ is wholly real, $A(n) = A(-n)^*$ and if $X(j)$ is wholly imaginary, then $A(n) = -A(-n)^*$.

Application of the DFT to real world data

There are several problems to be resolved when you attempt to apply the FFT algorithm to real world data to convert a time series to the frequency domain. The first step is to sample the data. It is easy to show that if a sinusoid of frequency $N/2+f$ is sampled, the sampled data is indistinguishable from a sinusoid of frequency $N/2-f$. This effect is called aliasing. To get unambiguous results, the input data must be band limited to the range 0 to N/2 (or in fact to any band of this width anywhere in the frequency spectrum). The frequency N/2 is usually referred to as the Nyquist critical frequency, or just the Nyquist frequency.

The next problem is that the maths assumes that the time series data is “circular”, that is that the frequency content is not changed in amplitude if the data is arbitrarily rotated. This assumption is, in general, not true, so when the discrete Fourier transform is used on real data the data is usually massaged in some way to make it circular. The usual method is to multiply the time series by some smooth function that is small at the ends and unity in the middle. The function is called a “window”, and the process is called “windowing”.

The good effect of a window is to reduce the spurious components caused by the discontinuity between the first and last sample. The bad effect is that the window causes each data point to be “smeared” in the result, and it significantly reduces the contribution of points near the ends of the sampled data to the result. The topic of windowing is complex and is beyond the scope of this document.

Finally, the results are for a discrete set of equally spaced frequencies. However, the input data will, in general not consist of these discrete frequencies. A sinusoid of an intermediate frequency will be represented in both adjacent discrete frequencies.

The CED FFT

Notice that the FFT transforms complex time series to complex frequency series. However, in the real world, sampled time series data does not normally include imaginary components. We usually want to transform wholly real time series to complex frequency series. There are three ways round this:

1. Set the imaginary components to 0 and calculate the FFT. This has the advantage of simplicity, but wastes a lot of time.

2. If you have two series to transform, treat one as real and the other as imaginary, do the transform, then use the symmetry properties of the DFT to separate out the data. This is fine if you have two arrays, but this is not always the case. This method also suffers from cross-spectral leakage between the two signals due to truncation and rounding inaccuracies in the calculations.
3. Split the data into two arrays and transform them together, then amalgamate the results and take advantage of the symmetry so that the results can be packed into the same space as the original.

We implement the third option, which seems complicated, but is actually no harder to program than the FFT itself. However, if you need to generate results including negative frequencies you have a small amount of post-processing work to do. As we do not have a complex data type available we store the real and imaginary components separately.

Forward FFT ArrFFT(x[], 1)

The forward transform takes N real data points and transforms them into N positive frequency complex components. The first component $A(0)$ is the mean amplitude of the original data points. The next $N/2$-1 components are the amplitudes of the real (cosine) components of data, the next component $A(N/2)$ is the Nyquist frequency amplitude, and the remaining $N/2$-1 components are the imaginary (sine) components.

If you want the results expressed in the same format as used by the DFT with negative frequencies, you will need more space to store the result. The example below assumes that we will transform 8 real data points:

DFT for 8 real data points

To transform 8 real data points, the DFT would require 8 real data points and 8 zeros for the imaginary components. In the transform, the real data (R_n) starts with the DC component, then three frequency components for positive frequencies, then the Nyquist component, followed by three components for negative frequencies. However, the negative frequency components are the same as the positive frequency components.

The imaginary result corresponding to DC is 0 (as all the imaginary data is zero). It is not quite so obvious why the imaginary Nyquist component is 0, but if you imagine a sine wave at the Nyquist frequency it would be zero at each sample. The imaginary frequency components (I_n) show odd symmetry around the DC and Nyquist frequency.

Real Imaginary (all zeros)

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	0	0	0	0	0	0	0	0

Transforms to

R_0	R_1	R_2	R_3	R_N	R_3	R_2	R_1	0	I_1	I_2	I_3	0	$-I_3$	$-I_2$	$-I_1$

ArrFFT(x[],1) equivalent

To do the same job with `ArrFFT()` you need supply only the 8 real data points. The real and imaginary results can also be fitted into 8 bins. Notice that the amplitudes of the components other than the DC and Nyquist are doubled compared to the DFT result above. You could argue that this is incorrect, but for a real problem, most users consider a sine wave at a frequency to be one positive frequency, not two sine waves of half the amplitude each, one at the positive frequency and one at the negative frequency.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7

Transforms to

R_0	$2R_1$	$2R_2$	$2R_3$	R_N	$2I_1$	$2I_2$	$2I_3$

If you really need the full result, it is a simple matter to expand the data into the full format. The R_n and I_n are the same as for the DFT above.

Inverse FFT ArrFFT(x[],2)

The inverse transform makes the assumption that the result will be purely real. To achieve this, the negative frequencies needed for the DFT are derived from the positive frequencies. The operation is the exact inverse of that described above.

A very common use of the inverse transform is to filter data by forward transforming, removing unwanted components, then inverse transforming. Another common use is to produce random signals with a known power spectrum by assigning known amplitudes and randomised phases and inverse transforming. In both these cases, the assumption that the output is wholly real is warranted.

Power and phase ArrFFT(x[],5)

The phase is calculated from the real and imaginary components with the phase of the DC and Nyquist components always 0, so they are omitted. The power is calculated so that (in terms of the DFT):

$$\frac{1}{N}\sum_{j=0}^{N-1}|X(j)|^2 = \sum_{n=0}^{N-1}|A(n)|^2$$

The `ArrFFT()` implementation starts with an array of real data and replaces it with an array of power components. The power components combine the negative and positive frequencies. In the diagram, P is power and Ph is phase. The sum of the five powers in the result is equal to the sum of the squares of the X_j divided by 8.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7

Transforms to

P_0	P_1	P_2	P_3	P_N	Ph_1	Ph_2	Ph_3

In terms of the result of the forward FFT described above:

$P_0 = R_0^2$, $P_1 = R_1^2 + I_1^2$, $P_2 = R_2^2 + I_2^2$, $P_3 = R_3^2 + I_3^2$, $P_N = R_N^2$

If you need to recover power for the positive and negative frequencies separately, the power at frequencies -3, -2 and -1 is $P_3/2$, $P_2/2$ and $P_1/2$ and the power for the positive frequencies is also halved. The phase given is correct for positive frequencies, however it must be changed to $2\pi\text{-}Ph_n$ for negative frequencies.

Summary

The `ArrFFT()` command presents results in a format that is convenient and easy to use for most practical uses of the FFT with real world data. However, if you need results that show both positive and negative frequencies, these are also easily obtained by simple manipulation of the results. Further, although not discussed here, you can generate a full complex to complex transform by using the `ArrFFT()` command twice.

References

You can find further information on the mathematics of the DFT and FFT in these references. The first is the famous paper that rediscovered the FFT. The second contains the algorithms for transforms of real only data. The third is a well-known book that covers the whole topic (and many others) in great depth.

1. Cooley, J. W., and Tukey, J. W., “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.,* vol. 19, pp. 297-301, April 1965.
2. Cooley, J. W., Lewis, P. A. W., and Welch, P.D., “Programming considerations in the calculation of sine, cosine and Laplace transforms,” *J. Sound Vib.*, vol 12, pp. 315-337, July 1970.
3. Rabiner, L. R., and Gold, B., 1975, *Theory and Application of Digital Signal Processing,* Pub. Prentice-Hall.

Correlation calculation

The waveform correlation in Spike2 often gives rise to questions about the compatibility of the results with those produced by systems that use Fast Fourier Transform (FFT) methods. The correlation in Spike2 was designed for use on arbitrarily long waveform sections that may contain discontinuous data. The length of the correlation was expected to be short compared to the number of data points in the original waveforms.

The FFT method is used to produce a mathematically efficient correlation between two continuous waveforms, which are usually a power of 2 points long. To use the FFT method, the data arrays must be zero padded for at least as many data points as the number of correlation points that are required. This is because the FFT assumes that the input data repeats cyclically, which is unlikely to be true in the general case.

The result of the waveform correlation in Spike2 is the same (allowing for floating point resolution) as using FFT methods on zero-padded data.

Consider the case of two continuous waveforms `a` and `b` that are both `Nw` points long. The correlation of these two waves at zero lag is:

```
Sumi(a[i]*b[i]) / Sqrt((Sumi(a[i]*a[i])*(Sumi(b[i]*b[i]))
```

where `Sumi(expr)` means sum the expression `expr` over all values of the index `i` over the `Nw` points. The top part of the expression is the cross product of the two arrays, the bottom part is a normalising function so that the result is 1.0 when the two arrays are identical. This produces a single output point. To produce the next output point, we must shift one of the arrays.

Method

Array `a` is the channel data, array `b` is the reference data.

Step 1: calculate the correlation with no shift

```
a   oooooooooooooooooooooooooooooooooo   Nw points long
b   oooooooooooooooooooooooooooooooooo   Nw points long
c   C.......                             Nc points long
```

Step 2: move array b one place right and calculate the correlation

```
a   oooooooooooooooooooooooooooooooooo   Nw-1 points correlated
b    oooooooooooooooooooooooooooooooooo  b array shifted right
c   CC......                             set second point
```

Step 3: move array b one place right and calculate the correlation

```
a   oooooooooooooooooooooooooooooooooo   Nw-2 points correlated
b     oooooooooooooooooooooooooooooooooo b array shifted right
c   CCC.....                             set third point
```

```
...
```

Step `Nc`: move array b one place right and calculate the correlation

```
a   oooooooooooooooooooooooooooooooooo
b          oooooooooooooooooooooooooooooooooo
c   CCCCCCCC                             set last point
```

Of course, this is a lot more complicated in Spike2 as we have to allow for time shifts between the two waveforms so that correlations at positive and negative times can be calculated. We also have to allow for gaps in the waveforms (as Spike2 allows waveform channels to have gaps). Further, we allow the user to add new waveform sections into the result and remove the effect of DC levels in the signals after calculating the correlation.

DC removal

If both signals have a non zero mean level, this mean level can dominate the result, giving an apparent large correlation at all time delays. If we wish to ignore the correlation due to mean levels we could do this by subtracting the mean from the signals before correlating them. However, in Spike2 we allow users to add additional data sections, so we would not know the mean level of the waveforms until all sections had been added in. To allow us to leave the mean level removal until all calculations are completed we observe that:

```
Sumi((a[i]-meana)*(b[i]-meanb)) =
                  Sumi(a[i]*b[i]) - Sumi(a[i]) * Sumi(b[i]) / n
```

where `meana` and `meanb` are the mean levels of both the waveforms and `n` is the total number of data points. From version 4.03 onwards, n is different for every result bin.

The result of this calculation will lose resolution the larger the mean levels become compared to the standard deviations of the waveforms. If the mean levels are large, you will get a much better result if the mean is removed from the original waveform data.

FFT method

The script `FFTCrl.s2s` is included with Spike2 from version 4. This contains the two procedures:

```
Proc AutoCrl(A[],norm%)
```

This replaces the array `A[]`, which must be a power of 2 points long, with the autocorrelation of `A[]` with itself. If `norm%` is non-zero, the result is normalised so that `A[0]` in the result has the value 1.

```
Proc CrossCrl(A[], B[], norm%)
```

This replaces the array `A[]` with the cross-correlation of `A[]` and `B[]`. The two arrays must have the same length and also be a power of 2 points long. If `norm%` is non-zero, the result is normalised so that `A[0]` would have the value 1 if the two arrays held identical data. `B[]` is not changed but cannot be the same array as `A[]`.

The first bin of the result is 0 lag, the next is a lag of one, and so on. However, the last bin is also a lag of -1, the next to last is a lag of -2 and so on. If you are interested in lags as large as +-`N`, you must extend your original data sets by `N` zeros. You also have the limitation that the input data must be a power of 2 in length, which means zero padding in the general case.

To use these routines to generate a correlation of length `Nc` from data arrays of length `Nw` you must start with arrays that are `Nw+Nc` rounded up to the next power of 2 in length. Set the first `Nw` bins to the data and fill the remainder of the array with zeros. You can now run the transform. The first `Nc` bins will hold the results for positive lags, the last `Nc-1` bins hold the results for negative lags.

Timing comparisons

The built-in Spike2 method takes a time proportional to the length of the original data times the number of correlation results. The time is given by:

$$T_s = k_s * Nw * Nc$$

The FFT method takes a time proportional to the number of points times the logarithm of the number of points. The time is given by:

$$T_f = k_f * N_f * Log_2(N_f)$$

`Nw+Nc` up to	Maximum `Nc`
1024	512
2048	338
4096	335
8192	348
16384	368
32768	390
65536	413
2^n	25.7n

N_f is `Nw+Nc` rounded up to the next power of 2. On my 450 MHz Pentium III and taking the times from 1000000 points, k_s is 1.4 10^{-8} seconds and k_f is 3.6 10^{-7} seconds. If you want a large number of lags and you have continuous data, you may prefer to use the FFT method. The table shows the values of `Nw` and `Nc` for which both methods take approximately the same time. If you make `Nc` substantially larger, the FFT method will be faster.

Put another way, if you want up to 400 points in the correlation result, the built-in method is probably the way to go. If you need substantially more result points, you will probably do better with the FFT method.

CED 1401-18 discriminator

Discriminator (CED 1401-18) support

The `Discrim…` family of commands supports the 1401-18 event discriminator card, which is available for the 1401*plus* only. You can also control the 1401-18 card interactively; see the Sample menu. This device is obsolete, so this documentation has been removed from the main script manual.

If you use these commands with the interactive discriminator dialog active, the dialog changes to show any changes made from the script. You cannot change the current channel the dialog displays (except with `DiscrimClear()`), so you will only see changes in the dialog if the channel is the same as the current channel in the dialog.

DiscrimChanGet()

This gets the input, spike2 event port, 1401 event input E1 or E3, mode, lower and higher trigger levels, and the time-out for modes 7 and 8 of a discriminator channel.

```
Func DiscrimChanGet(chan%, &in%, &out%, &mode%, &low, &hi, &tOut);
```

`chan%` the discriminator channel number (0-7).

`in%` Returned holding the source of the discriminator channel as:

0	The discriminator is not used
1	From the 1401 front panel digital input port
2	From the front panel event input (channels 0-4) or ADC Ext (channel 5)

`out%` Returned holding a code that describes how the discriminator output, event inputs and digital inputs are connected to the Spike2 event ports and the E1 (digital marker trigger) and E3 (start sampling trigger). The E1 and E3 values are only valid for channels 1 and 3.

	Spike2 event port	**E1 (digital marker) or E3 (start sampling)**
0	Disconnected.	Disconnected
1	1401 digital input	Disconnected
2	Discriminator output	Disconnected
4	Disconnected	To front panel E1 or E3
5	1401 digital input	To E1 or E3
6	Discriminator output	To E1 or E3
8	Disconnected	Discriminator output
9	1401 digital input	Discriminator output
10	Discriminator output	Discriminator output

`mode%` the mode of the discriminator channel is returned in this variable:

1	detect a level below a threshold
2	detect a level above a threshold
3	pulse on rising signal through a threshold
4	pulse on falling signal through a threshold
5	detect a level inside a region set by two thresholds
6	detect a level outside region set by two thresholds
7	pulse when signal returns below lower threshold, without going above the higher threshold, within a programmable time out period
8	pulse when signal returns above higher threshold, without going below the lower threshold, within a programmable time out period

`low` Returned holding the lower threshold level in Volts.

`high` Returned holding the higher threshold level in Volts.

`tOut` Returns the programmable time out period used in modes 7 and 8.

Returns 0 if all well or a negative error code.

See also:`DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimChanSet()

This sets the input, output, mode, lower and higher trigger levels, and the time-out (if the mode is 7 or 8) of the discriminator channel.

```
Func DiscrimChanSet(chan%, in%, out%{, mode%, low, high, tOut});
```

`chan%` the discriminator channel number (0-7).

`in%` Sets the source of the discriminator channel as:

0 Disconnected
1 From the 1401 front panel digital input port
2 From the front panel event input (channels 0-4) or ADC Ext (channel 5). For discriminator channels 6 and 7, you cannot set `in%` to 2 as there is no 1401 event input available as the source.

`out%` Sets how the discriminator output, event inputs and digital inputs connect to the Spike2 event ports and the E1 (digital marker) and E3 (start sampling) triggers. If `in%` is 0, `out%` is forced to 1. If both `in%` and `out%` are 1, `out%` is forced to 2. Possible `out%` values are:

0 Spike2 event port disconnected
1 Spike2 event port connected to 1401 digital input
2 Spike2 event port connected to discriminator output

When the channel is 1 or 3 you can choose what the 1401 event input E1 (digital marker) or E3 (start sampling) trigger connects to by adding 0, 4 or 8 to `out%`. If `in%` is 0, adding 4 is forced. If `in%` is 2, adding 4 is treated as adding 8.

0 E1 or E3 is disconnected
4 E1 or E3 is connected to the front panel E1 or E3
8 E1 or E3 is connected to the discriminator output

`mode%` The discriminator channel mode (1-8). If omitted, no change is made.

1 detect a level below a threshold
2 detect a level above a threshold
3 pulse on rising signal through a threshold
4 pulse on falling signal through a threshold
5 detect a level inside a region set by two thresholds
6 detect a level outside region set by two thresholds
7 pulse when signal returns below lower threshold, without going above the higher threshold, within a programmable time out period
8 pulse when signal returns above higher threshold, without going below the lower threshold, within a programmable time out period

`low` Sets the lower threshold level in volts between -5 Volts and the higher threshold level value. If omitted, no change is made. If the lower threshold level is given greater than the higher one, it will be set equal to the higher level.

`high` Sets the higher threshold level in Volts between the lower threshold level value and +5 Volts. If omitted, no change is made.

`tOut` Sets the time out period in seconds (0.00002 - 0.65535). This is only meaningful in mode 7 or 8. If omitted, no change is made.

Spike2 stores and uses the values for the threshold levels and time out period in terms of the resolution of the 1401-18 card. To find out the real value of the threshold levels and time out, use `DiscrimLevel()` and `DiscrimTimeOut()`. The difference is usually very small.

Returns 0 if the operation completed without a problem, or a negative error code.

See also: `DiscrimChanGet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimClear()

This sets the Discriminator configuration dialogue contents to a standard state. All the discriminator channels are disconnected. Spike2 event ports are connected to digital inputs. E1 (digital marker trigger) and E3 (start sampling trigger) are connected to the front panel E1 and E3. All the discriminator channels have mode 3, and the lower and higher trigger levels are set to 1.25 and 2.5 Volts. The ADC monitor channel is set to 15. If the discriminator dialog is open, channel 0 becomes the current channel.

```
Proc DiscrimClear();
```

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimLevel()

This sets or gets the lower or higher threshold level values of the discriminator channel.

```
Func DiscrimLevel(chan%, which%, {level});
```

`chan%`	The discriminator channel number (0-7).
`which%`	Selects which level to set or get: 0 = lower, 1 = upper threshold level.
level	If present it sets the threshold value in Volts: If `which%` is 0, it is the lower level. It should lie between -5 volts and the higher level. If it is set greater than the higher level, it is set equal to the higher one. If `which%` is 1, it is the higher level. It should lie between the lower level and 5 Volts. If it is set less than the lower level, it is set equal to the lower one
Returns	the lower or higher threshold level of the discriminator channel at the time of call, or a negative error code.

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimMode()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimMode()

This sets or gets the mode of the discriminator channel.

```
Func DiscrimMode(chan%, {mode%});
```

`chan%`	The discriminator channel number (0-7).
`mode%`	The mode of the discriminator channel (1-8). If omitted, the current mode of the discriminator channel is returned.
Returns	The discriminator channel mode at the time of call, or a negative error code.

1	detect a level below a threshold
2	detect a level above a threshold
3	pulse on rising signal through a threshold
4	pulse on falling signal through a threshold
5	detect a level inside a region set by two thresholds
6	detect a level outside region set by two thresholds
7	pulse when signal returns below lower threshold, without going above the higher threshold, within a programmable time out period
8	pulse when signal returns above higher threshold, without going below the lower threshold, within a programmable time out period

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMonitor()`, `DiscrimTimeOut()`

DiscrimMonitor()

This sets or gets the ADC monitor channel in the Discriminator configuration dialog.

```
Func DiscrimMonitor({chan%});
```

`chan%` Sets the ADC monitor channel number (0-15). If this is omitted, the current ADC monitor channel number is returned.

Returns the ADC monitor channel number at the time of call, or a negative error code.

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimTimeOut()`

DiscrimTimeOut()

This sets or gets the time out period of mode 7 or 8 for the discriminator channel.

```
Func DiscrimTimeOut(chan%, {tOut});
```

`chan%` the discriminator channel number (0-7).

`tOut` the programmable time out period in seconds (0.00002 - 0.65535). This is only meaningful in mode 7 or 8. If this is omitted, the current time out period is returned.

Returns the time out period at the time of call, or a negative error code.

See also: `DiscrimChanGet()`, `DiscrimChanSet()`, `DiscrimClear()`, `DiscrimLevel()`, `DiscrimMode()`, `DiscrimMonitor()`