WaveTrak XCMDs and XFCNs

General Considerations

As we saw in the previous chapter, the commands or functions that WaveTrak executes are written in both HyperTalk and as XCMDs/XFCNs written in C and assembly language. An XCMD (external command) or XFCN (external function) is nothing more than a module of executable code (a *code resource* in Mac terminology) that is called by a stack to perform an operation. The advantage of these externals is their execution speed. WaveTrak relies heavily on these to perform most of its real-time and computationally intensive operations. In fact, WaveTrak uses the HyperCard stack metaphor only as a familiar shell to interact with the user; most of the work is done by the more than 70 XCMDs. XCMDs and XFCNs are virtually identical, except that an XFCN returns a value (because it's a function) whereas an XCMD does not. The terms are used interchangeably in this manual when externals are referred to in the generic sense. The calling sequence is slightly different as well; XFCNs require parentheses around their parameters, whereas XCMDs do not:

Example 1:

```
get myXFCN (param1, param2, param3)

myXCMD param1,param2,param3
```

As far as you, the programmer, are concerned, XCMDs appear as standard HyperTalk commands, and XFCNs as standard HyperTalk functions. Unless otherwise stated, you must supply the required number of parameters, even if one of them is empty:

Example 2:

```
  put empty into param2
  myXCMD param1,param2,param3
```
or
```
  myXCMD param1,,param3  -- param2 is empty
```

When an external needs to return only a single value, it is usually created as an XFCN. However, many externals need to return several values; for example, the data acquisition commands must return up to 16 waves at a time. The way an external returns multiple values is by passing them back in global variables. *You inform an external of which globals to use by passing the* name(s) *of the global variables*:

Example 3:

```
  -- acquire a single wave
  -- return result in global variable 'theWave'
  global theWave
  ...
  AcqWave
sampleInterval,npoints,startMUX,endMUX,"theWave"
```

Example 4:

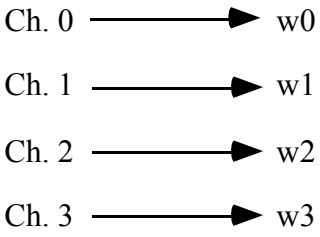
```
  -- acquire two waves
  global w0,w1
  ...
  AcqWave
sampleInterval,npoints,startMUX,endMUX,"w0,w1"
```
or
```
  global w0,w1
  ...
  put "w0,w1" into gList
  AcqWave sampleInterval,npoints,startMUX,endMUX,gList
```

It doesn't matter what globals you choose to receive the data, as long as there are at least as many globals as there are channels to be acquired. If you change the starting channel, but keep the *number* of channels the same, you don't have to change the globals that receive the new data, as long as you keep track what goes where. Here's an example:

Example 5:

```
  global w0,w1,w2,w3
  ...
  put "w0,w1,w2,w3" into gList
  put 0 into startMUX  -- acquire 4 channels
  put 3 into endMUX
  AcqWave sampleInterval,npoints,startMUX,endMUX,gList
```

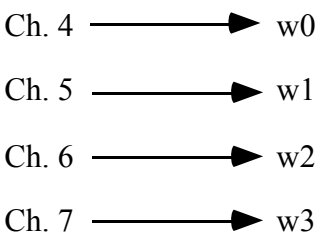
The data will be returned as follows:



changing the channels:

```
  global w0,w1,w2,w3
  ...
  put "w0,w1,w2,w3" into gList
  put 4 into startMUX  -- still 4 channels, but start
at #4
  put 7 into endMUX
  AcqWave sampleInterval,npoints,startMUX,endMUX,gList
```

The data will be returned as follows:



The description of the Multiple button in the Scripting chapter explains how to easily manage multiple channel acquisition using wave arrays.

Example 6:

```
  -- import a wave from the clipboard
  global theWave

  put "-12" into resultType  -- signed 12-bit wave
  -- translate scrap from X,Y into WTRK compressed
format
  -- result goes into global theWave
  get XYTableToWave (GetScrap(),"theWave",resultType)
  -- function returns info on conversion
  put it into XYresult
  put item 1 of XYresult into sampleInterval
  put item 2 of XYresult into npoints
```

GetScrap() returns as its value the contents of the clipboard, and is directly passed as the first parameter to XYTableToWave, which converts the ASCII data into a compressed WaveTrak wave, returning it in the global "theWave".

Tip:

As a general rule, acquisition commands and other externals that either return waves in globals (e.g. `AcqWave`) or require one or more waves as parameters (e.g. `AddWaves`) expect the *names* of the global variables containing or receiving the waves, that is, these globals are passed by *reference*. You can only pass *global* variables by reference (you cannot pass the name of a local variable and expect the XCMD to return data there). In contrast, externals operating on non-wave data usually expect the parameters to be passed by *value* (e.g. `CommaToTab`). You can therefore pass *global* or *local* variables to these functions. The description of each external clearly describes how parameters should be passed.

Acquisition Timing

A very important issue is the way acquisition commands digitize signals. In example 3, say you acquire a single wave at 50 µs/sample (`sampleInterval` = 50). This

straightforward acquisition is illustrated in Fig. 11-1A. The A/D converter will "freeze" the analog value on the sample-and-hold circuit and convert this analog level to a digital value. The conversion process, along with software overhead needed to read and store successive conversions, requires about 7 µs/sample, therefore the minimum sampling interval for the MacADIOS II on-board A/D converter is 7. There is also an upper limit of 13107 µs for the sampling interval.

Technical note:

Timer channel number 5 on the AM9513 chip is used for clocking A/D conversions. The timers have 16 bits of resolution and are clocked at a frequency of 5 MHz, therefore the maximum sample interval is 65535/5 MHz = 13107 µs.

Because the MacADIOS II hardware has only one sample-and-hold amplifier, sampling multiple channels simultaneously cannot be performed in true synchrony. WaveTrak attempts to make the conversions as close to synchronous as possible, by sampling successive channels as soon as the previous channel's conversion is complete. Fig. 11-1B illustrates the case for two channels again sampled at 50 µs/sample (`sampleInterval` = 50, example 4). Note that WaveTrak acquisition commands are designed to conserve the number of points and sampling rate regardless of the number of channels requested (within the limitations of the A/D converter of course). So in example 4, when two channels are requested with the same sample interval of 50 µs, the converter actually must perform conversions at an average rate of 25 µs/conversion. If you requested 8 channels at 50 µs/channel, an error would result because the hardware would have to perform conversions at an average rate of about 6 µs/conversion, which is beyond its capability. In practice, sampling two or more channels adds another microsecond to the conversion process to allow the multiplexer time to switch. Higher gains (x100), may require even more time (8-9 µs/conversion) for the circuitry to settle or you may end up with noisy data.

Tip:

As a general rule, the minimum sample interval is 7 µs (142 kHz) when acquiring a single channel. For two or more channels, you must ensure that the *effective* sample interval (i.e. sampleInterval parameter ÷ number of channels) be 8 µs (125 kHz) or longer.

Example:

Script demonstrating how the sample interval must be scaled when multiple channels are being digitized.

Correct

```
-- 1 channel, 7µs: OK
put 7 into sampleInterval
put 1 into startMUX
put 1 into endMUX
AcqWave sampleInterval,npoints,startMUX,endMUX,gList
```

or

```
-- 2 channels, 8µs/channel: OK
put 16 into sampleInterval
put 1 into startMUX
put 2 into endMUX
AcqWave sampleInterval,npoints,startMUX,endMUX,gList
```

Incorrect

```
-- 2 channels would require a 4µs converter: Error
put 8 into sampleInterval
put 1 into startMUX
put 2 into endMUX
AcqWave sampleInterval,npoints,startMUX,endMUX,gList
```

Normally, the 7 µs lag between adjacent channels does not cause significant problems, and you can manipulate the waves as if they were acquired synchronously. However, if you sample 16 channels for example, the actual time between point 0 from channel 0 and point

0 from channel 15 will be about 105 µs (15 * 7 µs), which may be unacceptable. Therefore, if you need for signals to be sampled as synchronously as possible, minimize time lags by feeding them to adjacent channels.

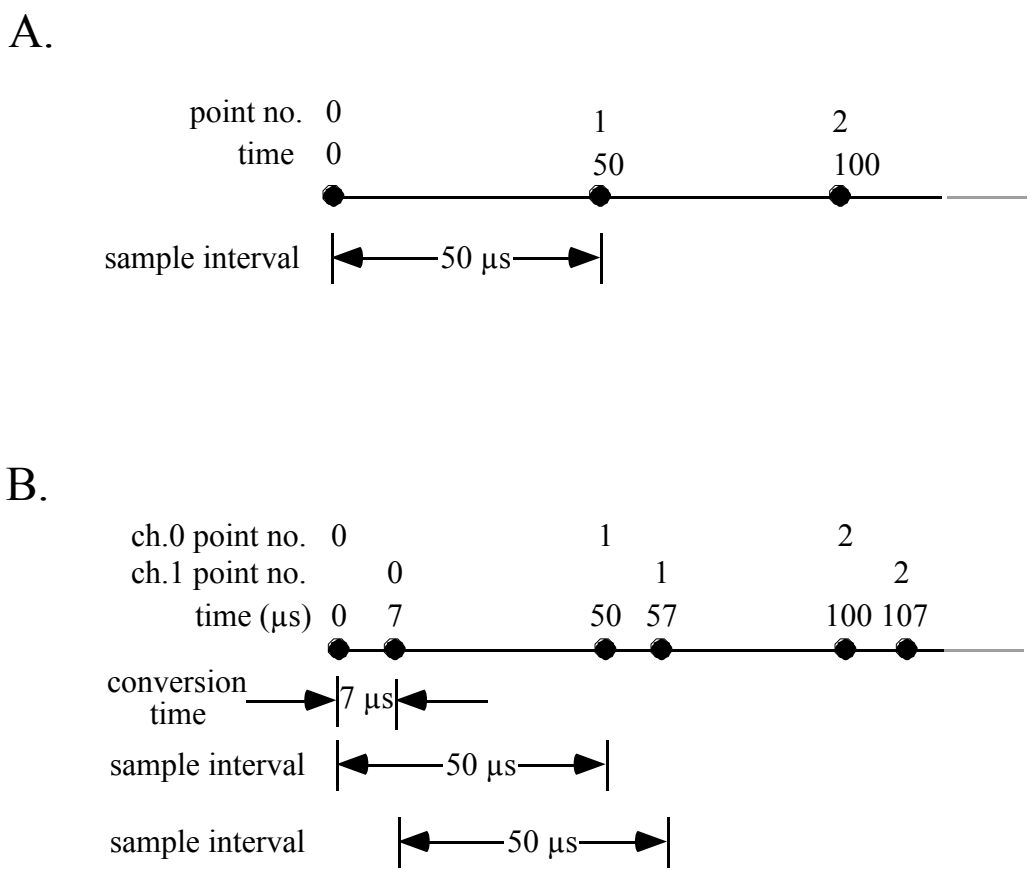


Fig. 11-1: Timing of A/D conversions for a single (A) and multiple (B) A/D channels. See text for details.

XCMDs and XFCNs inform you of any errors by placing an error number in the global variable **XCMDErr**. A value of zero means no errors occurred. A later chapter lists all WaveTrak errors along with descriptions of their causes. It's a good idea to check the value of **XCMDErr** after every call to an XCMD to make sure it executed successfully.

This chapter lists all of WaveTrak's XCMDs and XFCNs in alphabetical order. In order to help you find the command you need, Tables 1 through 5 list all the externals by functional category along with a brief description.

Table 1: Data acquisition *(WaveTrak AD version only)*.

AcqMean	Measures the mean analog level at the A/D input(s). Used to measure the baseline DC level of a channel before triggering an event.
AcqWave	Digitizes from 1 to 16 A/D channels simultaneously. The number of samples is limited by the amount of RAM.
AcqWaveTTL	Digitizes from 1 to 16 A/D channels while generating a (delayed) pulse at one of the TTL outputs. Used to stimulate a system with a TTL pulse while measuring its response.
AcqWaveTimer	Digitizes from 1 to 16 A/D channels while generating a (delayed) pulse at one of the AM9513 timer chip outputs. Allows for very accurate pulse delay/width generation.
AcqWaveDAC	Digitizes from 1 to 16 A/D channels while generating a (delayed) pulse at one of the D/A outputs. Used to stimulate a system with an analog pulse while measuring its response.
AcqWaveOnTTL	Begins acquisition on either a rising or falling edge at one of the TTL inputs.
AcqWavePreTTL	Implements a pre-triggering function, where part of the acquisition is captured before an edge occurs at the TTL input port. It is frequently necessary to know what happened immediately before some event.
AcqWaveThresh	Begins acquisition when the signal crosses a pre-determined analog threshold, analogous to an oscilloscope trigger level.
AvgWaveTTL	Averages a series of waveforms while generating (delayed) pulses at one of the TTL outputs.
AvgWaveTimer	Averages a series of waveforms while generating (delayed) pulses at one of the timer/counter chip outputs.

ReadTTLbit	Reads value of one TTL input bit.
ReadTTLbyte	Reads all 8 TTL input bits.

Table 2: Signal generation and related functions
(WaveTrak AD version only).

SetADGain	Sets the on-board programmable gain amplifier.
DACPulse	Generates an analog pulse at the D/A converter.
WriteDAC	Writes an analog value to D/A converter.
StartPulseTrain	Program the timer/counter chip to output a continuous pulse train at a given frequency and pulse width. Signal will continue after this XFCN returns.
StopPulseTrain	Stops counter/timer (e.g. stops a StartPulseTrain command).
WriteModeByte	Writes a value to the MacADIOS II mode register.
WriteTTLbit	Sets/clears one TTL output bit.
WriteTTLbyte	Writes a byte to all 8 bits of TTL output port.

Many of the wave math functions have a `resultType` parameter which allows you to select what data type the operation will return. Standard WaveTrak data types are discussed in detail in the Scripting chapter.

Table 3: Waveform math, analysis and digital signal processing.

AddWaveK	Adds a constant to a wave. Used to offset waves for plotting, or to remove DC component.
AddWaves	Adds two waves together.
SubtractWaves	Subtracts two waves.
MultWaveK	Multiplies a wave by a constant. Used to numerically amplify/attenuate waves.
MultWaves	Multiplies two waves.
DivideWaves	Divides two waves.
InitWaveK	Initializes a wave to a constant.
InitWaveSin	Initializes a wave to a sine function.
InitWaveNoise	Initializes a wave to white noise.
GetWaveStats	Computes wave statistics such as mean, StdDev, RMS value, min, max, range, etc.
MeanWave	Computes mean value of a segment of a wave. Used to compute baseline of a segment preceding a stimulus, or to get the DC level of a signal.
AreaWave	Computes the area under a wave.
AbsAreaWave	Computes the absolute (rectified) area under a wave.
ConvolveWave	Convolves a wave with a set of FIR filter coefficients to implement the filter.
DesignFIRhi	Computes the coefficients of a hi-pass FIR filter.
DesignFIRlo	Computes the coefficients of a lo-pass FIR filter.
DifferentiateWave	Differentiates a wave.
IntegrateWave	Integrates a wave.
ThresholdWave	Threshold detects a wave against a preset trigger level.
AmplSpectrum	Computes the frequency (amplitude)

	spectrum of a wave with the FFT.
PowerSpectrum	Computes the frequency (power) spectrum of a wave with the FFT.
FilterWaveFFTloLog	Lo-pass filters a wave using the FFT (log rolloff).
FilterWaveFFThiLog	Hi-pass filters a wave using the FFT (log rolloff).
FilterWaveFFTloLin	Lo-pass filters a wave using the FFT (linear rolloff).
FilterWaveFFThiLin	Hi-pass filters a wave using the FFT (linear rolloff).
FilterWaveFIR	Implements a finite impulse response (FIR) non-recursive digital filter.
Window	Multiplies a wave by a window function (e.g. Hanning, Parzen, Welch) prior to spectrum estimation or filtering to reduce leakage.
Trim	Deletes segments of a wave. Used to remove unwanted segments after a response or preceding a stimulus; or to remove segments digitized during a long conditioning pulse generated by e.g. `AcqWaveDAC`; or to isolate a segment for analysis.
WaveToEventList	Generates a list of events (zero-crossings) from a wave.

Table 4: Drawing, importing and exporting.

DrawWave	Draws up to 16 waves simultaneously on the screen.
DrawWaveCoords	Draws up to 16 waves simultaneously on the screen; supports X,Y cursor read-out, zoom in or out for detailed inspection.
OverlayWave	Overlays up to 16 waves in a window, without erasing existing waves.
CopyXYTable	Converts a wave to numerical X,Y values in text format, and copies to clipboard. Used for exporting digitized waves into a spreadsheet or statistical program.
CopyYTable	Converts a wave to numerical Y values in text format, and copies to clipboard.
CopyPICT	Converts wave into a 72 dpi graphics object for pasting into a drawing program like Canvas or MacDraw.
CopyBigPICT	Converts wave into a graphics object for pasting into a drawing program like Canvas or MacDraw. Wave is converted into a large polygon, so that it can be reduced to maintain full LaserWriter resolution.
WaveToXYTable	Converts a wave to numerical X,Y values in text format.
WaveToYTable	Converts a wave to numerical Y values in text format.
XYTableToWave	Converts an ASCII table of XY or Y data to a wave.
CommaToTab	Converts a variable from comma- to tab-delimited format.
TabToComma	Converts a variable from tab- to comma-delimited format.
ScrapToComma	Converts contents of clipboard from tab- to comma-delimited format.
ScrapToTab	Converts contents of clipboard from comma- to tab-delimited format.

Table 5: Miscellaneous functions.

CheckFPU	Checks for presence of 68881/882 chip. Used to disable functions (such as FFTs & filters) that require FPU support.
HardwareInit *(A/D version only)*	Initializes and checks hardware, checks for presence of MacADIOS II card.
GetScrap	Pastes the contents of the clipboard into a HyperCard variable.
PutScrap	Copies a HyperCard variable to the clipboard. Used to export data generated with a HyperCard script for pasting into another application.
PutToGlobal	Places a value into a global. Useful for manipulating arrays of globals.

Alphabetical Listing of XCMDs and XFCNs

The following pages contain detailed descriptions of the XCMDs and XFCNs in WaveTrak's data acquisition and signal processing toolbox. Note that XCMDs and XFCNs that perform data acquistion require the optional WaveTrak AD version of the software. Tables 1 and 2 above list list the functions that require the AD version and a MacADIOS II data acquisition board.

You will refer to this section frequently when modifying existing scripts or writing your own. 'Type' tells you whether the external is an XCMD or XFCN. 'Syntax' illustrates a typical call; note that longer lines will wrap in the manual but must be typed on a single line in the script editor, or broken up with the option-return character (¬). A 'Description' follows, explaining what the external does and how to use it. 'Result' summarizes the data returned by the command, and examples illustrate its use.

AbsAreaWave

Type: XFCN

Syntax:
```
AbsAreaWave
("theWave",sampleInterval,startTime,endTime,baseline)
```

Description:
Computes the absolute (rectified) area (values below baseline are reflected above before sum) of samples between `startTime` and `endTime` μs included, with respect to `baseline`. Mathematically:

$$\text{sampleInterval} \sum_{\nu=0}^{\nu\pi o\iota\nu\tau\sigma-1} \left| \psi_{\nu} - \beta\alpha\sigma\varepsilon\lambda\iota \right.$$

where:

npoints is the number of points in the wave
y_n is the value of the nth point
`baseline` and `sampleInterval` are parameters

`"theWave"` is the *name* of the global containing the wave (double quotes are included to remind you to pass the wave by name, not by value). `sampleInterval` is the original sampling interval in μs, `startTime` and `endTime` define the segment of the wave to be summed. Passing -1 in `endTime` tells the XFCN to continue to the last point. For example, pass `startTime` = 0,`endTime` = -1 to compute area of entire wave, or `startTime`,`endTime` = -1 to compute area from `startTime` to end of wave.

Result:
Returns a real value with dimensions μs `x` vertical unit, formatted according to **XYCoordYformat** global. This is a raw area i.e. a sum of integer or floating point values, which must be corrected for full-scale range, A/D coding, amplifier gain, etc...

Example:
This example is taken from the 'Plot Abs Area' button in the root cards. It is assumed that a wave has been previously stored in a trace card along with its parameters in the 'HParams' field:

```
-- the original sampling rate
put line 3 in bg fld "HParams" into sampleInterval
-- copy the previously acquired wave into 'theWave'
put bg fld "data" into theWave
-- what's the data type?
put getWaveType(theWave) into resultType
-- get the previously measured baseline
put line 2 in bg fld "Readings" into baseline
-- convert baseline from mV to binary
get line 4 in bg fld "HParams"  -- full scale and units
put item 1 in it into bottomY
put item 2 in it into topY
put translateToBinary(baseline,bottomY,topY,resultType)
into binBase
-- now compute the absolute area, result goes into 'it'
get
AbsAreaWave("theWave",sampleInterval,startTime,endTime,
binBase)
```

AcqMean

Type: XCMD

Syntax:
`AcqMean sampleInterval,npoints,startMUX,endMUX,gList`

Description:
Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel, and computes mean of each channel. `gList` contains a comma-delimited list of global names where means will be returned. Useful for measuring DC baselines before an event, or averaging a series of samples (rather than a single reading) for a more reliable DC reading.

Result:
Returns real values formatted according to **XYCoordYformat** global. These are raw means which must be corrected for full-scale range, A/D coding, amplifier gain, etc...

Example:
```
global sampleInterval,npoints,FSTable,theMean,ADCbits
put 0 into startMUX       -- the A/D channel
put startMUX into endMUX  -- just one channel
AcqMean
sampleInterval,npoints,startMUX,endMUX,"theMean"
-- translate into mV
put item 1 in line (startMUX+1) in FSTable into
minFullScale
put item 2 in line (startMUX+1) in FSTable into
maxFullScale
get
translateToReal(theMean,minFullScale,maxFullScale,ADCbi
ts)
```

AcqWave

Type: XCMD

Syntax:
`AcqWave sampleInterval,npoints,startMUX,endMUX,gList`

Description:
The simplest of the signal acquisition XCMDs. Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. See Fig. 11-1 and the discussion on acquisition timing and sampling multiple channels earlier in this chapter.

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`.

Example:
```
global sampleInterval,npoints,FSTable
-- globals to receive digitized data
global w0,w1,w2,w3,w4,w5,w6,w7

put 0 into startMUX
put 7 into endMUX
-- put global NAMES into a list
put "w0,w1,w2,w3,w4,w5,w6,w7" into gList
put endMux-startMUX+1 into nChannels
-- adjust sample interval for number of channels
put sampleInterval*nChannels into effectiveSint
AcqWave effectiveSint,npoints,startMUX,endMUX,gList
```

AcqWaveDAC

Type: XCMD

Syntax:
`AcqWaveDAC sampleInterval,npoints,startMUX,endMUX, DACchannel,DACpre,DACpulse,DACpost,prePulse,pulseWidth, gList`

Description:
Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. Simultaneously steps one D/A channel (`DACchannel`: 0 or 1) to one or two analog levels as shown in Fig. 11-2. Analog levels (`DACpre`, `DACpulse`, `DACpost`) are in mV. `prePulse` and `pulseWidth` are in µs and must be positive and *integral multiples* of `sampleInterval`. If `prePulse` is zero, only a single pulse will be generated. The `DACpost` analog level will persist after the XCMD returns, until a new level is written to that D/A channel. The total pre-pulse and pulse time must be less than the length of the acquisition (i.e. `prePulse+pulseWidth < sampleInterval*npoints`). Needs globals **DACmin**, **DACmax** and **DACbits** to automatically translate mV to a valid binary count (these are initialized at start-up). You cannot generate a pre-pulse and delay the start of the acquisition until the onset of the pulse. Instead, acquire the entire signal, then use the 'Trim' command to remove the unwanted segment acquired during the pre-pulse time.

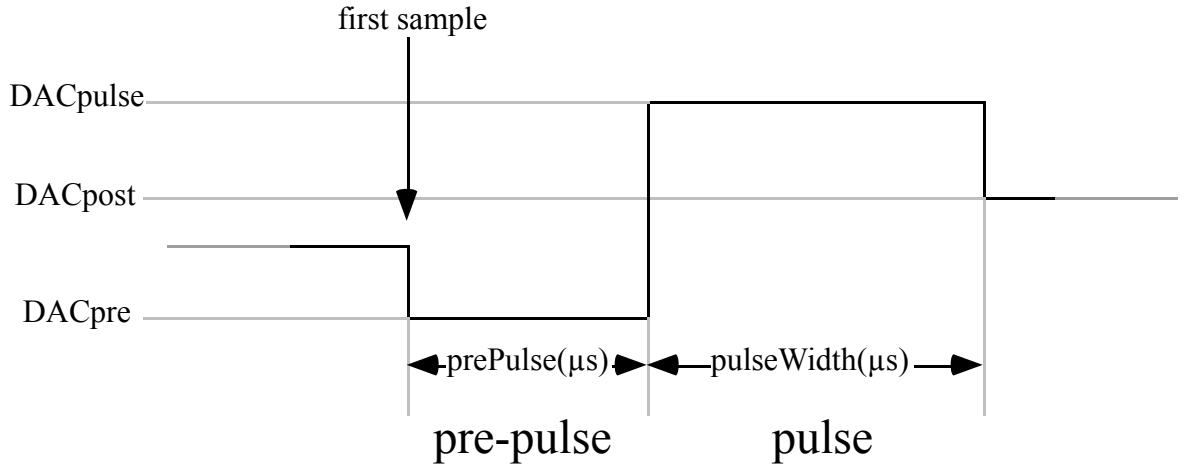


Fig. 11-2: Relationship between analog pulses and sample acquisition.

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`.

Example:

```
global sampleInterval,npoints,FSTable,DACGainTable
global theWave

put 0 into DACchannel     -- the D/A channel, 0 or 1
put -5000 into DACpre     -- analog level of pre-pulse (in mV)
put 500 into prePulse     -- duration of pre-pulse (in µs)
put 7000 into DACpulse    -- analog level during pulse (in mV)
put 500 into pulseWidth   -- duration of pulseWidth (in µs)
put 0 into DACpost        -- final analog level after pulse (in mV)
put 0 into startMUX       -- the selected A/D channel
put startMUX into endMUX  -- a single channel only

-- adjust pulseWidth and prePulse to multiples of sampleInterval
put round(pulseWidth/sampleInterval)*sampleInterval into pulseWidth
put round(prePulse/sampleInterval)*sampleInterval into prePulse
-- adjust DAC levels w.r.t external DAC gain
get line (DACchannel+1) in DACGainTable
put round(DACpre/it) into adjDACpre
put round(DACpulse/it) into adjDACpulse
put round(DACpost/it) into adjDACpost
-- acquire the wave
AcqWaveDAC sampleInterval, npoints, startMUX, endMUX,¬
DACchannel,adjDACpre,adjDACpulse,adjDACpost,¬
prePulse,pulseWidth,"theWave"
```

AcqWaveOnTTL

Type: XCMD

Syntax:
```
AcqWaveOnTTL sampleInterval,npoints,startMUX,endMUX,
TTLbit,edge,timeout,gList
```

Description:
Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. Begins acquisition on a lo-to-hi (`edge` = 1) or hi-to-lo (`edge` = 0) transition of bit `TTLbit` of digital input port. Returns with silent **XCMDErr** = 17 if required edge was not detected within approximately `timeout` (integer: 1 to 800) seconds. Used to synchronize acquisition with an external TTL trigger or event detector.

Trigger uncertainty with respect to the TTL edge: ≈ 2µs on a Mac II, ≈1.3 µs on Mac IIci, ≈0.8 µs on Mac IIfx. Trigger pulse must be at least as wide as the uncertainty or the edge might be missed.
The Mac will be completely locked out during the time the XCMD waits for an edge (for a maximum of `timeout` seconds).

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`.

Example:

```
global XCMDErr
global sampleInterval,npoints
global theWave
put 0 into TTLbit
put 0 into edge     -- trigger on rising (1) or falling
(0) edge
put 2 into timeout  -- quit if no edge detected after 2
sec
put 0 into startMUX       -- the A/D channel
put startMUX into endMUX  -- a single channel only
-- acquire the wave
AcqWaveOnTTL sampleInterval,npoints,startMUX,endMUX,¬
TTLbit,edge,timeout,"theWave"
```

AcqWavePreTTL

Type: XCMD

Syntax:
```
AcqWavePreTTL sampleInterval,npoints,startMUX,endMUX,
TTLbit,edge,preTrig,timeout,gList
```

Description:
Similar to `AcqWaveOnTTL` but implements pre-triggering, capturing a segment *before* the edge transition. Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. Captures a segment `preTrig` µs long before a lo-to-hi (`edge` = 1) or hi-to-lo (`edge` = 0) transition of bit `TTLbit` of digital input port. `preTrig` must be an integral multiple of `sampleInterval`. Returns with silent **XCMDErr** = 17 if required edge was not detected within approximately `timeout` (integer: 1 to 800) seconds. Used as a pre-trigger function to capture signals preceding a trigger or event.

Trigger uncertainty: ≈ 2µs on a Mac II, ≈1.3 µs on Mac IIci, ≈0.8 µs on Mac IIfx. Trigger pulse must be at least as wide as uncertainty or edge might be undetected.

The Mac will be completely locked out during the time the XCMD waits for an edge (for a maximum of `timeout` seconds).

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`.

Example:

```
global XCMDErr
global sampleInterval,npoints
global theWave
put 0 into TTLbit
put 0 into edge  -- trigger on rising (1) or falling
(0) edge
put 2 into timeout   -- quit if no edge detected after
2 sec
put 500 into preTrig -- acquire this many µs before
trigger edge
put 0 into startMUX       -- the A/D channel
put startMUX into endMUX  -- a single channel only
-- adjust preTrig to multiple of sampleInterval
put round(preTrig/sampleInterval)*sampleInterval into
preTrig
-- acquire the wave
AcqWavePreTTL
sampleInterval,npoints,startMUX,endMUX,TTLbit,edge,¬
preTrig,timeout,"theWave"
```

AcqWaveThresh

Type: XCMD

Syntax:
```
AcqWaveThresh sampleInterval,npoints,startMUX,endMUX,
thresh,slope,timeout,gList
```

Description:
Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. Begins acquisition when signal at A/D channel `startMUX` crosses analog threshold `thresh` (binary count) (`slope` = 1 for positive crossing, `slope` = 0 for negative crossing). Hysteresis of 50 integer counts is built in to avoid false triggering when signal crosses threshold with opposite slope. Returns with silent **XCMDErr** = 17 if threshold crossing was not detected within approximately `timeout` (integer: 1 to 800) seconds. Useful for triggering on a predetermined point on a signal, similar to the trigger level of an oscilloscope.

Threshold must be translated to a binary count with respect to full-scale range and binary coding (see example). The Mac will be completely locked out during the time the XCMD waits for an edge (for a maximum of `timeout` seconds).

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`.

Example:

```
global sampleInterval,npoints,FSTable
global theWave,ADCbits
put -500 into triggerLevel  -- threshold in mV
put 1 into slope    -- +ve (1) or -ve (0) threshold
crossing
put 1 into timeout  -- time out limit in seconds
put 0 into startMUX -- the selected channel
put startMUX into endMUX
-- translate threshold to binary count
put item 1 in line (startMUX+1) in FSTable into
minFullScale
put item 2 in line (startMUX+1) in FSTable into
maxFullScale
put translateToBinary(triggerLevel,minFullScale,¬
maxFullScale,ADCbits) into binThresh
AcqWaveThresh sampleInterval,npoints,startMUX,¬
endMUX,binThresh,slope,timeout,"theWave"
```

AcqWaveTimer

Type: XFCN

Syntax:
`AcqWaveTimer (sampleInterval,npoints,startMUX,endMUX, timerChannel,preTrig,pulseWidth,gList)`

Description:
Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. A positive TTL pulse is generated at one of the counter/timer chip outputs (`timerChannel`: 1 to 4), `preTrig` µs after beginning acquisition; pulse lasts for `pulseWidth` µs as shown in Fig. 11-3. `preTrig` and `pulseWidth` are in µs, must be positive integers, but need *not* be multiples of `sampleInterval` (see `AcqWaveTTL` in contrast). If `preTrig` is zero, pulse onset will coincide with first sample. `preTrig` must be less than the length of the acquisition (i.e. `preTrig` < `sampleInterval*npoints`), but pulse width can extend beyond sample window. If `pulseWidth` = 0, no pulse is generated and acquisition proceeds as in `AcqWave`. Used to trigger an external device at the same time or just after starting the acquisition.

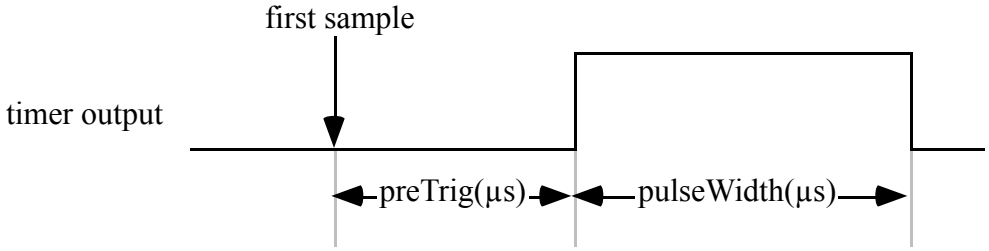


Fig. 11-3: Relationship between digital pulse and sample acquisition.

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`. Because the AM9513 timer chip has 16 bit counters, the resolution of the pre-triggering time and pulse width may be limited. The result of this XFCN contains three comma delimited items representing the actual preTrig and pulseWidth times that were generated, and the possible resolution for these times given the present parameters (all in

µs). If the actual parameters differ from those passed in the parameter list, a silent error 60 is returned in **XCMDErr** global.

Example 1:

```
global sampleInterval,npoints,theWave
put 1 into timerChannel   -- timer channel generating pulse (1-4)
put 500 into preTrig      -- segment acquired before pulse (in µs)
put 1000 into pulseWidth  -- duration of pulseWidth (in µs)
put 0 into startMUX       -- the selected channel
put startMUX into endMUX  -- a single channel only
put AcqWaveTimer (sampleInterval,npoints,startMUX,endMUX,¬
timerChannel,preTrig,pulseWidth,"theWave")
```

This code segment would write '500,1000,1' in the message box, and **XCMDErr** would be 0 indicating that the actual pre-trigger time and pulse width were exactly as you requested. Furthermore, item 3 in the result tells you that you can set each of these two parameters with a resolution of 1 µs.

Example 2:

```
global sampleInterval,npoints,theWave
put 1 into timerChannel    -- timer channel generating pulse (1-4)
put 48993 into preTrig     -- segment acquired before pulse (in µs)
put 100002 into pulseWidth -- duration of pulseWidth (in µs)
put 0 into startMUX        -- the selected channel
put startMUX into endMUX   -- a single channel only
put AcqWaveTimer (sampleInterval,npoints,startMUX,endMUX,¬
timerChannel,preTrig,pulseWidth,"theWave")
```

This example would put '48994,100002,2' in the message box, and **XCMDErr** would be 60 indicating that the possible resolution for such long times was 2 µs. The actual pre-trigger time was 48994 (vs. the requested 48993), but the pulse width was accurate. As you can see the percent errors are very small, and the timer chip will generate pulses with an accuracy of ±200 ns of that reported by the value of the XFCN.

AcqWaveTTL

Type: XCMD

Syntax:
```
AcqWaveTTL sampleInterval,npoints,startMUX,endMUX,
TTLbit,preTrig,pulseWidth,gList
```

Description:
Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. Toggles one TTL output bit (`TTLbit`: 0 to 7) `preTrig` µs after beginning acquisition; toggles same bit again after `pulseWidth` µs as shown in Fig. 11-4. `preTrig` and `pulseWidth` are in µs and must be positive and integral multiples of `sampleInterval`. If `preTrig` is zero, pulse onset will coincide with first sample. The total pre-triggering and pulse time must be less than the length of the acquisition (i.e. `preTrig+pulseWidth < sampleInterval*npoints`).

Used to trigger an external device at the same time or just after starting the acquisition. Because the output bit is *toggled*, the polarity of the TTL pulse is determined by the starting level; use the 'WriteTTLbit' command to preset the bit level. We recommend using `AcqWaveTimer` instead, unless you have reason not to.

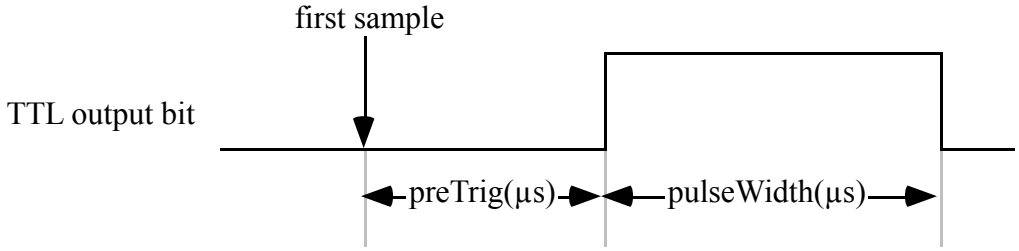


Fig. 11-4: Relationship between digital pulse and sample acquisition.

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`.

Example:

```
global sampleInterval,npoints,FSTable,theWave
put 0 into TTLbit     -- which TTL output do you want to pulse (0-7)
put 500 into preTrig     -- segment acquired before pulse (in µs)
put 1000 into pulseWidth -- duration of pulseWidth (in µs)
put 0 into startMUX      -- the selected channel
put startMUX into endMUX -- a single channel only
-- adjust pulseWidth and preTrig to multiples of sampleInterval
put round(pulseWidth/sampleInterval)*sampleInterval into pulseWidth
put round(preTrig/sampleInterval)*sampleInterval into preTrig
-- acquire the wave
AcqWaveTTL sampleInterval,npoints,startMUX,endMUX,TTLbit,preTrig,¬
pulseWidth,"theWave"
```

AddWaveK

Type: XFCN

Syntax:
`AddWaveK ("theWave",K,resultType)`

Description:
Adds a constant `K` (need not be an integer) to every point in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global). `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Useful for offsetting waves or removing DC components. Passing `K` = 0 is useful for changing a wave from one data type to another. Pass a negative value in `K` to subtract a constant.

Result:
Returns compressed wave as value of XFCN.

Example:
```
global theWave
put 100 into K
put 0 into resultType
-- add 100 to every point in theWave, return same data
type
put AddWaveK ("theWave",K,resultType) into theWave
put 0 into K
put "F" into resultType
-- convert theWave to a floating point type
put AddWaveK ("theWave",K,resultType) into theWave
```

AddWaves

Type: XFCN

Syntax:
```
AddWaves ("wave1","wave2",resultType)
```

Description:
Adds two waves together, point by point:

$$y_n = y_{n,wave1} + y_{n,wave2}$$

If number of points in both waves is different, stops summing when reaches the end of the shorter wave. Double quotes are included to remind you to pass the *names* of the globals containing the waves. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `wave1`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

Result:
Returns compressed wave as value of XFCN.

Example:
```
global wave1,wave2
put 0 into resultType
-- add wave1 and wave2, result goes in wave1
put AddWaves ("wave1","wave2",resultType) into wave1
put "F" into resultType
-- add wave1 and wave2, floating point result goes in theWave
-- note that theWave need not be a global
put AddWaves ("wave1","wave2",resultType) into theWave
```

AmplSpectrum

Type: XFCN

Syntax:
`AmplSpectrum ("theWave",dB,floor)`

Description:
Computes frequency (amplitude) spectrum of wave in global `theWave` (double quotes are included to remind you to pass the *name* of the global containing the wave):

$$y_n = \sqrt{Re_n^2 + Im_n^2}$$

where:

y_n = nth frequency component
Re_n = real part of nth element after FFT
Im_n = imaginary part of nth element after FFT

The number of points in `theWave` must be an integral power of 2 for the FFT. If `db` = TRUE, converts spectrum to a log scale and returns elements in dB normalized to maximum value (= 0 dB). Values less than `floor` will be clipped to `floor`; this is to avoid very large negative components with a log scale. If `floor` = 0, small values are not clipped and 0 elements in spectrum (which should be -∞ on a log scale) are returned as -3.403E+38 (the smallest single precision floating point number, because HyperCard does not recognize the -INF symbol). dB values are computed as follows:

$$dB = 20 \log \left(\frac{y_n}{y_{max}} \right)$$

Result:
Returns compressed wave as value of XFCN. Result is always a floating point wave (type "F").

Example:

```
global w0,theWave
put TRUE into dB   -- display on a log scale,
normalized to 0 dB
put -80 into floor -- clip very small components to -80
dB
put AmplSpectrum ("theWave",dB,floor) into w0
```

AreaWave

Type: XFCN

Syntax:
```
AreaWave
("theWave",sampleInterval,startTime,endTime,baseline)
```

Description:
Computes the net area under wave, between `startTime` and `endTime` µs included, with respect to `baseline`. Mathematically:

$$\text{sampleInterval} \sum_{\nu=0}^{\nu\pi o\iota\nu\tau\sigma-1} \psi_{\nu} \; - \; \beta\alpha\sigma\varepsilon\lambda$$

where:

npoints is the number of points in the wave
y_n is the value of the nth point
`baseline` and `sampleInterval` are parameters

`"theWave"` is the *name* of the global containing the wave (double quotes are included to remind you to pass the wave by name, not by value). `sampleInterval` is the original sampling interval in µs, `startTime` and `endTime` define the segment of the wave to be summed. Passing -1 in `endTime` tells the XFCN to continue to the last point. For example, pass `startTime` = 0,`endTime` = -1 to compute area of entire wave, or `startTime`,`endTime` = -1 to compute area from `startTime` to end of wave.

Result:
Returns a real value with dimensions µs `x` vertical unit, formatted according to **XYCoordYformat** global. This is a raw area i.e. a sum of integer or floating point values, which must be corrected for full-scale range, A/D coding, amplifier gain, etc...

Example:
This example assumes that a wave has been previously stored in a trace card along with its parameters in the 'HParams' field:

```
-- the original sampling rate
put line 3 in bg fld "HParams" into sampleInterval
-- copy the previously acquired wave into 'theWave'
put bg fld "data" into theWave
-- what's the data type?
put getWaveType(theWave) into resultType
-- copy the previously measured baseline
put line 2 in bg fld "Readings" into baseline
-- convert baseline from mV to binary
get line 4 in bg fld "HParams"  -- full scale and units
put item 1 in it into bottomY
put item 2 in it into topY
put translateToBinary(baseline,bottomY,topY,resultType)
into binBase
-- now compute the area, result goes into 'it'
get
AreaWave("theWave",sampleInterval,startTime,endTime,bin
Base)
```

AvgWaveTimer

Type: XFCN

Syntax:
`AvgWaveTimer (sampleInterval,npoints,startMUX,endMUX, timerChannel,preTrig,pulseWidth,gList,nAvg,period,lock)`

Description:
This XFCN averages a number acquisitions. Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. A positive TTL pulse is generated at one of the counter/timer chip outputs (`timerChannel`: 1 to 3), `preTrig` µs after beginning acquisition; pulse lasts for `pulseWidth` µs as shown in Fig. 11-5. `preTrig` and `pulseWidth` are in µs, must be positive integers, but *need not be* multiples of `sampleInterval` (see `AcqWaveTTL` in contrast). If `preTrig` is zero, pulse onset will coincide with first sample. `preTrig` must be less than the length of each acquisition (i.e. `preTrig` < `sampleInterval*npoints`), but pulse width can extend beyond sample window. If `pulseWidth` = 0, no pulse is generated. `nAvg` (must be ≤ 32767) cycles are averaged with a period of `period` µs. If `lock` is TRUE, then interrupts are disabled and the Mac will be locked out for the entire averaging period. Period jitter will be about 1 µs, as measured on a Mac IIfx. If `lock` is FALSE then acquisition can be aborted prematurely with command-period (a silent error 64 is returned in **XCMDErr**), and waves already acquired will be returned correctly. However, period can jitter by as much as ±500 µs. Set `lock` to TRUE if you need very precise timing of the period.

Timer channels 4 and 5 are used to time the cycles and A/D sampling. There is a software overhead between cycles used to accumulate the most recent wave and reset the timers. It can be estimated as ≈ `npoints`*3.5µs + 70µs on a Mac IIfx with `lock` = TRUE, and ≈ `npoints`*3.5µs + 300µs with `lock` = FALSE. Slower Macs will have proportionally greater overheads depending on their clock speed. Therefore, the minimum reliable period is ≥ `npoints*sampleInterval` (sample window time) + `npoints`*3.5µs + 70µs (overhead) with `lock` = TRUE. The XFCN will *not* signal an error if you violate the overhead, so be careful, or your data may be inaccurate.
Use this XFCN to trigger an external device at the same time or just after starting

the acquisition.

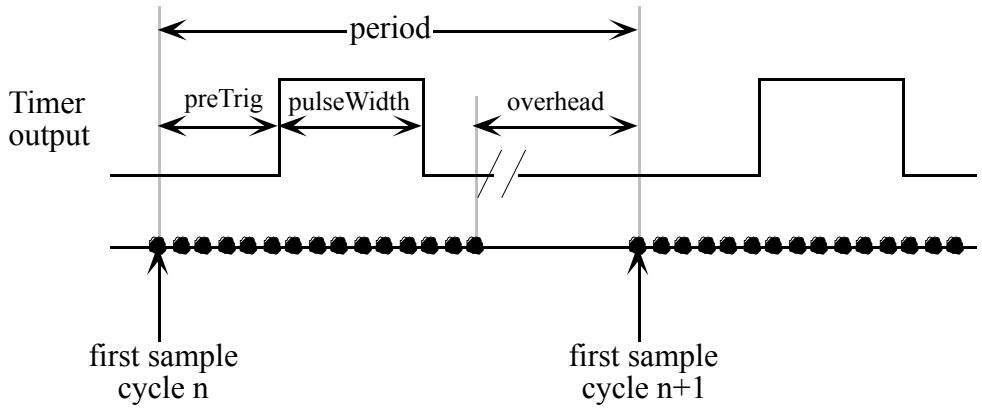


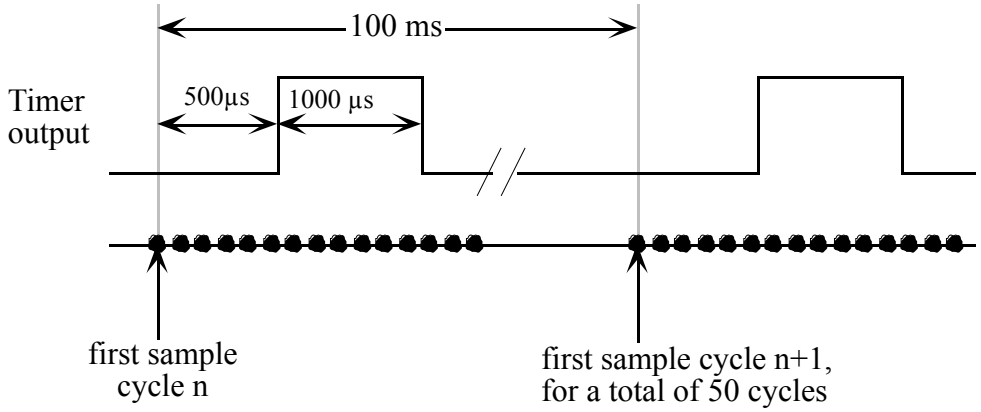
Fig. 11-5: Relationship between digital pulses, sample acquisition and averaging period. There is a software overhead between averaging cycles.

Result:
Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`. Because the AM9513 timer chip has 16 bit counters, the resolution of the pre-triggering time and pulse width may be limited. The result of this XFCN contains two lines (delimited by carriage returns): the first contains a single item representing the actual number of cycles averaged. This may be less than `nAvg` if user typed command-period to abort. The second line contains three comma delimited items representing the actual preTrig and pulseWidth times that were generated, and the possible resolution for these times given the present parameters (all in μs). If the actual parameters differ from those passed in the parameter list, a silent error 60 is returned in **XCMDErr** global.

Example:

```
global sampleInterval,npoints,theWave
put 1 into timerChannel  -- timer channel generating pulse (1-3)
put 500 into preTrig     -- segment acquired before pulse (in µs)
put 1000 into pulseWidth -- duration of pulseWidth (in µs)
put 50 into nAvg         -- number of waves to be averaged
put 100000 into period   -- averaging period (in µs)
put FALSE into lock      -- allow cmd-period to abort the run
put 0 into startMUX      -- the selected A/D channel
put startMUX into endMUX -- a single channel only
get AvgWaveTimer (sampleInterval,npoints,startMUX,endMUX,¬
timerChannel,preTrig,pulseWidth,"theWave",nAvg,period,lock)
```

This code segment will average 50 waves from A/D channel 0, at 100 ms intervals. The timing would be as follows:



The variable `it` will contain (assuming the user did not press command-period):

```
50
500,1000,1
```

and **XCMDErr** would be 0 indicating that 50 cycles were averaged, and the actual pre-trigger time and pulseWidth were exactly as requested. Furthermore, the result tells you that you can set each of these two parameters with a resolution of 1 µs. See `AcqWaveTimer` for an example of what happens when a pulse could not be generated exactly as you requested.

AvgWaveTTL

Type: XFCN

Syntax:
```
AvgWaveTTL
(sampleInterval,npoints,startMUX,endMUX,TTLbit,
preTrig,pulseWidth,gList,nAvg,period,lock)
```

Description:
This XFCN averages a number acquisitions. Reads `npoints` samples from each A/D channel (`startMUX` to `endMUX` inclusive) at a sample rate of `sampleInterval` µs/channel. `gList` contains a comma-delimited list of global names where waves will be returned. Toggles one TTL output bit (`TTLbit`: 0 to 7) `preTrig` µs after beginning acquisition; toggles same bit again after `pulseWidth` µs as shown in Fig. 11-6. `preTrig` and `pulseWidth` are in µs and must be positive and integral multiples of `sampleInterval`. If `preTrig` is zero, pulse onset will coincide with first sample. The total pre-triggering and pulse time must be less than the length of the acquisition (i.e. `preTrig+pulseWidth` < `sampleInterval*npoints`). If `pulseWidth` = 0, no pulse is generated. `nAvg` (must be ≤ 32767) cycles are averaged with a period of `period` µs. Because timer channel 4 is used to measure the time between cycles, `period` need not be a multiple of `sampleInterval`. If `lock` is TRUE, then interrupts are disabled and the Mac will be locked out for the entire averaging period. Period jitter is ≈ 1µs measured on a Mac IIfx. If `lock` is FALSE then acquisition can be aborted prematurely with command-period (a silent error 64 is returned in **XCMDErr**), and waves already acquired will be returned correctly. However, period can jitter by as much as ±500 µs. Set `lock` to TRUE if you need very precise timing of the period.

Timer channels 4 and 5 are used to time the cycles and A/D sampling. There is a software overhead between cycles used to accumulate the most recent wave and reset the timers. It can be estimated as ≈ `npoints`*3.5µs + 70µs on a Mac IIfx with `lock` = TRUE, and ≈ `npoints`*3.5µs + 300µs with `lock` = FALSE. Slower Macs will have proportionally greater overheads depending on their clock speed. Therefore, the minimum reliable period is ≥ `npoints*sampleInterval` (sample window time) + `npoints`*3.5µs + 70µs (overhead) with `lock` = TRUE. The XFCN will *not* signal an error if you violate

the overhead, so be careful, or your data may be inaccurate.
Because the output bit is *toggled*, the polarity of the TTL pulse is determined by the starting level; use the '`WriteTTLbit`' command to preset the bit level. We recommend using `AvgWaveTimer` instead, unless you have reason not to.

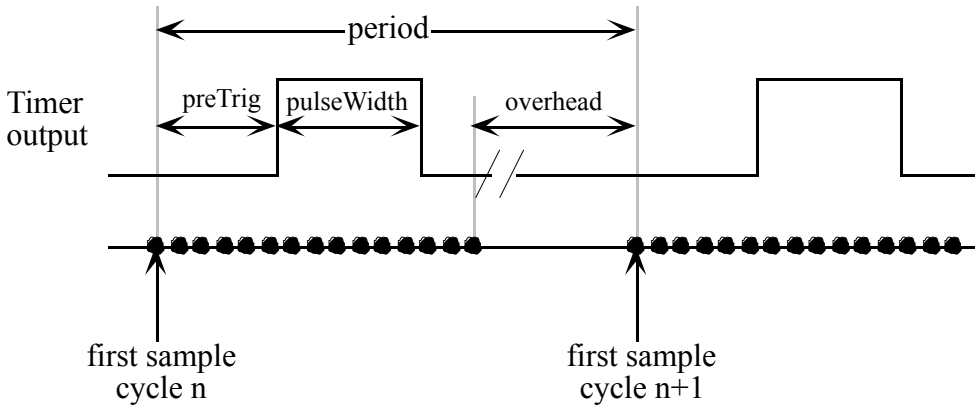


Fig. 11-6: Relationship between digital pulses, sample acquisition and averaging period.

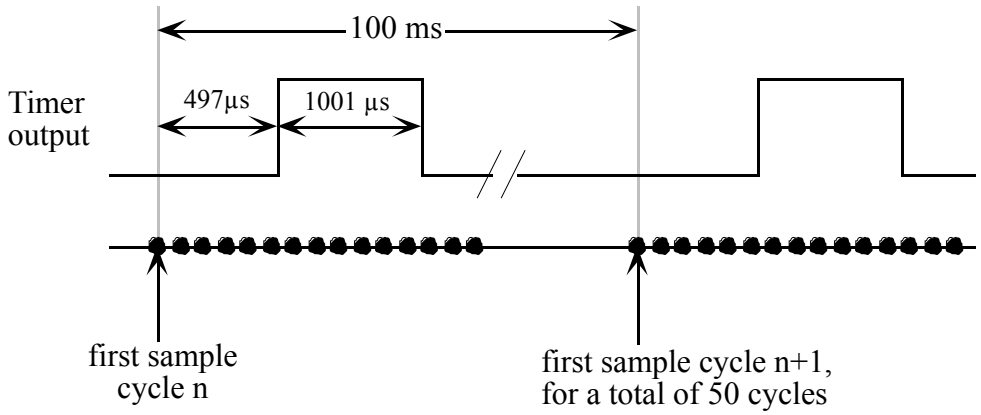
Result:

Returns compressed waves in globals whose *names* are passed as a comma-delimited list in `gList`. The result of this XFCN contains the actual number of cycles averaged. This may be less than `nAvg` if user typed command-period to abort.

Example:

```
global npoints,theWave
put 7 into sampleInterval
put 0 into TTLbit    -- which TTL output do you want to pulse (0-7)
put 500 into preTrig     -- segment acquired before pulse (in µs)
put 1000 into pulseWidth -- duration of pulseWidth (in µs)
put 50 into nAvg         -- number of waves to be averaged
put 100000 into period   -- averaging period (in µs)
put FALSE into lock      -- allow cmd-period to abort the run
put 0 into startMUX      -- the selected channel
put startMUX into endMUX -- a single channel only
-- adjust pulseWidth and preTrig to multiples of sampleInterval
put round(pulseWidth/sampleInterval)*sampleInterval into pulseWidth
put round(preTrig/sampleInterval)*sampleInterval into preTrig
put AvgWaveTTL (sampleInterval,npoints,startMUX,endMUX,¬
TTLbit,preTrig,pulseWidth,"theWave",nAvg,period,lock)
```

This code segment will average 50 waves from A/D channel 0, at 100 ms intervals. The timing would be as follows (note that the pre-triggering time and pulse width had to be rounded to the nearest multiple of `sampleInterval`):



The message box will read 50 (assuming the user did not press command-period).

CheckFPU

Type: XFCN

Syntax:
```
CheckFPU ()
```

Description:
This XFCN checks if the Mac has a 68881/882 floating point unit. Use this XFCN to check for a math coprocessor before calling a routine that requires one.

> Technical note:
> This XFCN calls the `SysEnvirons` toolbox trap to check if the machine has an FPU.

Result:
Returns TRUE if a 68881/882 floating point unit is installed, otherwise it returns FALSE.

CommaToTab

Type: XFCN

Syntax:
`CommaToTab (theData)`

Description:
This XFCN converts a variable (`theData`) from comma-delimited to tab-delimited format. Note that the variable `theData` is passed by *value* (no quotes), and not by name.
Use this XFCN to convert HyperCard lists and tables, which are comma-delimited, to standard Mac export format (tab-delimited) for pasting into spreadsheets, etc. You will commonly call `PutScrap` next to copy the contents to the clipboard.

Result:
Returns as the value of the function, the data with all commas changed to tab characters. Existing tabs are left unchanged.

Example:
```
-- make a table
put "1,2,3" & return into x
put "4,5,6" & return after x
put CommaToTab(x) into x  -- convert to tab-delimited
format
PutScrap x                -- copy to clipboard
```

`x` is passed to `CommaToTab` by value, therefore it is not enclosed in double quotes. The contents of the clipboard will be:

```
    1    2    3
    4    5    6
```

where each item in a row is separated by a tab character.

ConvolveWave

Type: XFCN

Syntax:
```
ConvolveWave ("theWave", "FIRcoeffs", resultType)
```

Description:
Implements a finite impulse response (non-recursive) digital filter. The wave to be filtered is passed in global `theWave`, and the coefficients describing the filter are passed in global `FIRCoeffs` (double quotes are included to remind you to pass the *names* of the globals). Unlike FFT-based filters, the number of points in the wave need not be an integral power of two. This XFCN differs from FilterWaveFIR in that it accepts filter coefficients from DesignFIRlo or DesignFIRhi XFCNs, rather than a complete set of coefficients generated externally. Use ConvolveWave only with filters generated by DesignFIRlo and DesignFIRhi (see the description for details).

`resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

Result:
Returns filtered, compressed wave as the value of the XFCN. The filtered wave is shifted left by n/2 samples (where n is the number of filter coefficients) so that the original and filtered waves are in phase (FIR filtering inherently shifts the output right by n/2 samples). Also, the first and last n/2 points are returned unfiltered, because there are n/2 points missing before the start of the wave, and after the end of the wave, required to compute the output.

Example:
You have a digitized signal sampled at 100 kHz and you want to remove high frequency noise by low-pass filtering the wave at 12 kHz. The filter must roll off so that all components are attenuated to less than 100 dB above 35 kHz. You must first design the filter by calling `DesignFIRlo` with the appropriate parameters, then filter your signal using `ConvolveWave`:

```
global theWave,FIRCoeffs,w0
put 0 into resultType      -- return same type of wave
-- filter the wave, and return result in same global
put 12000 into fpass  -- in Hz
put 35000 into fstop
put 100 into dB
put 10 into sampleInterval  -- 100kHz is 10µs sample
interval
-- design filter coeffs
put DesignFIRlo(fpass,fstop,dB,sampleInterval) into
FIRCoeffs
-- filter it, result into global w0
put ConvolveWave ("theWave", "FIRCoeffs", resultType)
into w0
```

DesignFIRlo or DesignFIRhi, and ConvolveWave are the most convenient and efficient (in terms of computation speed) way to filter signals in WaveTrak.

CopyBigPICT

Type: XFCN

Syntax:
```
CopyBigPICT
("theWave",nVertex,leftX,rightX,topY,bottomY,
baseline,Xcal,Ycal,Xunit,Yunit)
```

Description:
This XFCN converts the wave stored in global `theWave` into a large, high-resolution PICT (graphics object) and copies it to the clipboard. The number of points in the resulting wave will be `nVertex`. `nVertex` will be limited to the number of points in the wave or 4096, whichever is less.
`leftX,rightX,topY,bottomY` define the wave in real units; see the discussion on wave descriptors in the Scripting chapter. These parameters are used to properly scale the calibration marks, which are generated as an L-shaped polygon, `Ycal` units high and `Xcal` units wide. The units of `Xcal` must be the same as `leftX` and `rightX`; similarly, `Ycal` is defined in the same units as `topY` and `bottomY`. The actual names of the units (must be < 32 characters in length each) are passed in `Xunit` (e.g. "µs") and `Yunit` (e.g. "mV"), and will be drawn next to the calibration marks. Pass zero to one or both calibration parameters to suppress that limb of the mark. If `bottomY` ≤ `baseline` ≤ `topY`, a horizontal baseline is drawn for reference. Pass a value beyond the `bottomY-topY` range to suppress it.
Waves of type float are treated somewhat differently. They are always translated into a PICT 4096 points tall, and scaled so that the maximum and minimum values in the wave will span this size. The `topY` and `bottomY` parameters are used only to place the baseline and scale the calibration marks.

The resulting large graphic can be scaled down in a object-oriented graphics program to the required size, and it will maintain full resolution when printed. The menu item 'Copy as Big PICT' uses this XFCN; see the chapter on WaveTrak menus for a complete description of how to export waves at high resolution.

Technical Note:

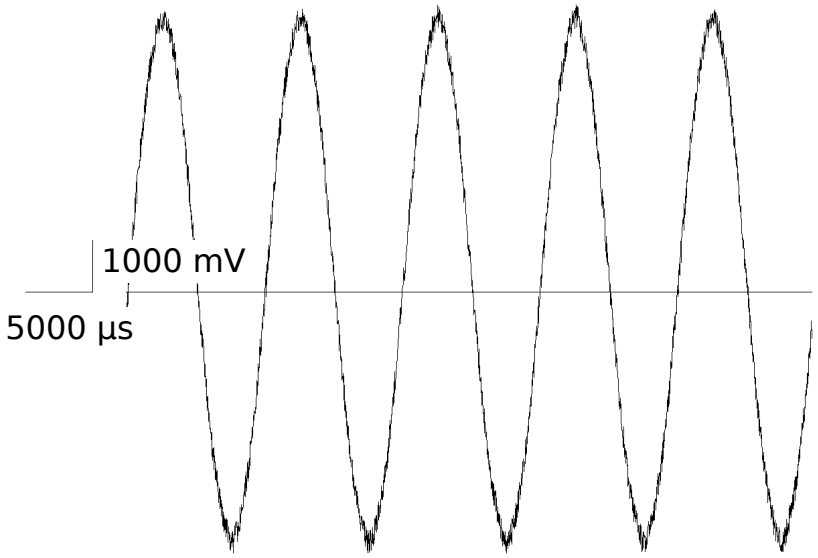
`CopyBigPICT` translates the wave into a QuickDraw polygon containing `nvertex` vertices. The maximum size of the polygon is 4096 QuickDraw points, which, when scaled down, gives you control of every dot on a page at the 300 dpi LaserWriter resolution. If `nVertex` > 400, the polygon is split up into several contiguous polygons, each consisting of 400 points to avoid generating a PostScript error.

Result:
Translates the wave into a large graphics object and copies it to the clipboard. The value of the XFCN is the actual number of vertices in the polygon, which may be less than `nVertex`.

Example:
This is an example of what you would get if you translated the sample sine wave in the first trace card using `CopyBigPICT`, pasted it into Canvas 2.1, and scaled it down. The code example assumes that all wave descriptors have been initialized when you opened the trace card:

```
global theWave,leftX, rightX, topY, bottomY, baseline
put 4096 into nVertex  -- max no. of vertices/wave
put 5000 into Xcal     -- 5000 µs = 5 ms cal mark
put 1000 into Ycal     -- 1000 mV = 1 volt cal mark
put "µs" into Xunit
put "mV" into Yunit
get CopyBigPICT ("theWave",nVertex,leftX,rightX,topY,bottomY,¬
baseline,Xcal,Ycal,Xunit,Yunit)
put it                 -- result to message box
```

Pasting into Canvas and scaling down:



Note that maximum resolution is maintained, and the lines are drawn as hairlines because the pen size was scaled down as well (a small amount of noise was added to better illustrate the high resolution capability of this XFCN). The message box will read 2048, the number of points in the original wave.

CopyPICT

Type: XCMD

Syntax:
```
CopyPICT "theWave",boundRect,leftX,rightX,topY,bottomY,
baseline,Xcal,Ycal,Xunit,Yunit
```

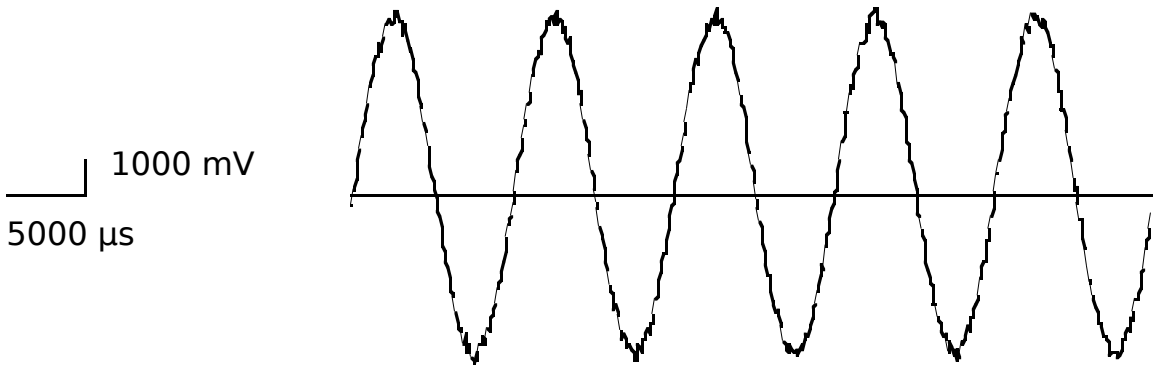
Description:
This XCMD converts the wave stored in global `theWave` into a low resolution 72 dpi PICT and copies it to the clipboard. The size of the wave in the PICT is determined by the rectangle, `boundRect`. You can pass any rectangle in `boundRect` to determine the final frame for your wave. The actual size of the PICT will be horizontally larger than `boundRect` by the size of the calibration marks. `leftX,rightX,topY,bottomY` define the wave in real units; see the discussion on wave descriptors in the Scripting chapter. These parameters are used to properly scale the calibration marks, which are generated as an L-shaped polygon, `Ycal` units high and `Xcal` units wide. The units of `Xcal` must be the same as `leftX` and `rightX`; similarly, `Ycal` is defined in the same units as `topY` and `bottomY`. The actual names of the units (must be < 32 characters in length each) are passed in `Xunit` (e.g. "µs") and `Yunit` (e.g. "mV"), and will be drawn next to the calibration marks. Pass zero to one or both calibration parameters to suppress that limb of the mark. If `bottomY` ≤ `baseline` ≤ `topY`, a horizontal baseline is drawn for reference. Pass a value beyond the `bottomY-topY` range to suppress it.
Waves of type float are treated somewhat differently. The PICT is scaled so that the actual maximum and minimum points in the wave will span the height of `boundRect`. The `topY` and `bottomY` parameters are used only to place the baseline and scale the calibration marks.

The menu item 'Copy as PICT' uses this XFCN; see the chapter on WaveTrak menus for a complete description of how to export waves using this XCMD. Use this XCMD to generate low resolution representations of your waves for viewing on the screen, or for rough drafts.

Result:
Translates the wave into a 72 dpi graphics object and copies it to the clipboard.

Example:
This is an example of what you would get if you translated the sample sine wave in the first trace card using `CopyPICT`. The code fragment assumes that all wave descriptors have been initialized when you opened the trace card:

```
global theWave,leftX, rightX, topY, bottomY, baseline
put "0,0,300,250" into boundRect  --
left,top,right,bottom
put 5000 into Xcal  -- 5000 µs = 5 ms cal mark
put 1000 into Ycal  -- 1000 mV = 1 volt cal mark
put "µs" into Xunit
put "mV" into Yunit
CopyPICT
"theWave",boundRect,leftX,rightX,topY,bottomY,¬
baseline,Xcal,Ycal,Xunit,Yunit
```



The same amount of noise was added as in the example for `CopyBigPICT` to illustrate the lower resolution of this export mode. `boundRect` defines a rectangle 250 points high, but this 12 bit wave is only 133 points high. The discrepancy is due to the fact that `boundRect` maps the *full scale* integer wave onto the given rectangle. Therefore, only if this 12-bit signed integer wave had elements spanning the full -2048 to +2047 range, would its size be that of `boundRect`, i.e. 250 points high. The present example does not span the full 12 bit range, and so the PICT is proportionally smaller (float waves, on the other hand, will always be scaled to full height because by definition, there is no full scale limit for this data type). The horizontal dimension, however, will always be the size of the horizontal dimension of `boundRect`. Regardless of the data type or final size, the calibration marks will always reflect the true dimensions of the wave.

CopyXYTable

Type: XCMD

Syntax:
```
CopyXYTable "theWave",leftX,rightX,topY,bottomY
```

Description:
This XCMD converts the wave stored in global `theWave` into a tab-delimited ASCII table and copies it to the clipboard. Both X and Y values are converted and copied. The X values are linearly mapped from their point numbers such that the first X value will be `leftX` and the last will be `rightX`. The **XYCoordXformat** global determines the format of the real X values. If you pass zero for both `leftX` and `rightX`, X values will simply be point numbers (i.e. 0, 1, 2 . . npoints-1). The conversion format will default to "%.0f" i.e. integers with no digits after the decimal point (see the chapter on WaveTrak globals for an explanation of conversion formats specified by **XYCoordXformat** and **XYCoordYformat** globals). Y values of integer waves are linearly mapped such that the maximum value (e.g. 2047 for a signed 12 bit wave) will be `topY` and minimum value (e.g. -2048) will be `bottomY`.

If you pass zero for both `topY` and `bottomY`, integer Y values are converted without scaling with a conversion format of "%.0f" (i.e. you will get integer values ranging from -2048 to 2047 for a signed 12 bit wave). Because there are no full scale limits for float waves, Y values are always converted without translation. The **XYCoordYformat** global determines the format of the real Y values.

Use this XCMD to export X-Y numerical data for pasting into a spreadsheet or wave processor like Igor.

Result:
Translates the wave into a tab-delimited table of X-Y data pairs and copies it to the clipboard.

Example:
This is an example of what you would get if you translated the sample sine wave in the first trace card. The code fragment assumes that all wave descriptors have been initialized when you opened the trace card:

```
global theWave,leftX, rightX, topY, bottomY
global XYCoordXformat,XYCoordYformat
-- save globals
put XYCoordXformat into tempX
put XYCoordYformat into tempY
-- 3 digits after the decimal point
put "%.3f" into XYCoordXformat
put "%.3f" into XYCoordYformat
CopyXYtable
"theWave",leftX/1000,rightX/1000,topY/1000,bottomY/1000
-- restore globals
put tempX into XYCoordXformat
put tempY into XYCoordYformat
```

Note that we elected to convert the X-Y values in ms and volts, rather than µs and mV, by passing `leftX/1000`, `rightX/1000`, `topY/1000`, `bottomY/1000` instead of `leftX`, `rightX`, `topY`, `bottomY`. To make sure that we get enough precision, the conversion formats were reset to 3 digits after the decimal point (`"%.3f"`). Note that we saved, then *restored* ,the values of **XYCoordXformat** and **XYCoordYformat** globals to avoid interfering with other WaveTrak functions. The clipboard will contain the following table:

```
  0.000  -0.364
  0.025  -0.291
  0.050  -0.208
    .
    .
    .
  51.150 -0.696
  51.175 -0.603
```

CopyYTable

Type: XCMD

Syntax:
```
CopyYTable "theWave",topY,bottomY
```

Description:
This XCMD functions exactly like `CopyXYTable`, except that it only copies a single column of Y values to the clipboard. `CopyYTable` converts the wave stored in global `theWave` (double quotes are included to remind you to pass the *name* of the global) into an ASCII table. Only Y values are converted. Y values of integer waves are linearly mapped such that the maximum value (e.g. 2047 for a signed 12 bit wave) will be `topY` and minimum value (e.g. -2048) will be `bottomY`. If you pass zero for both `topY` and `bottomY`, integer Y values are converted without scaling with a conversion format of "%.0f" (i.e. you will get integer values ranging from -2048 to 2047 for a signed 12 bit wave). Because there are no full scale limits for float waves, Y values are converted without translation. The **XYCoordYformat** global determines the format of the real Y values.

Use this XCMD to export numerical data for pasting into a spreadsheet or wave processor like Igor. If you are exporting several waves with the same number of points and sample interval, use `CopyXYTable` for the first one, to get a column of X values, then use `CopyYTable` for subsequent waves, to export only Y values and avoid duplicating X values.

Result:
Translates the wave into a column of Y data, and copies it to the clipboard.

Example:
This is an example of what you would get if you translated the sample sine wave in the first trace card. The code fragment assumes that all wave descriptors have been initialized when you opened the trace card:

```
global theWave
CopyYTable "theWave",0,0
```

Here we elected to convert the Y values as raw integers without scaling, by passing zero for both `topY` and `bottomY`. Any conversion format in the **XYCoordYformat** global was ignored and defaulted to "%.0f". The clipboard will contain the following column of data:

```
-75
-60
-43
  .
  .
  .
-143
-124
```

DACPulse

Type: XFCN

Syntax:
```
DACPulse
(DACchannel,DACpulse,pulseWidth,DACpost,TTLbit)
```

Description:
Generates a single analog pulse at one D/A channel (`DACchannel`: 0 or 1). Analog voltage will be `DACpulse` mV during pulse and will last `pulseWidth` µs. The D/A output is stepped back to `DACpost` mV after the pulse. If you have any external gain or attenuation, `DACpulse` and `DACpost` must be adjusted accordingly (see example below). Frequently it is convenient to trigger another device such as a scope simultaneously with the pulse. The digital output bit `TTLbit` (0 to 7) will be toggled for the duration of the analog pulse. Pass -1 in `TTLbit`, or omit this parameter altogether, to suppress digital pulse generation.

> Technical note:
>
> This function uses the AM9513 timer number 4 to time the width of the pulse. Because the AM9513 has 16 bit counters, the resolution of longer pulses may be limited (see example). All Macintosh interrupts are disabled and the Mac is locked during the pulse.

Result:
The value of the XFCN returns a comma-delimited list containing the actual width of the pulse (which may differ slightly from `pulseWidth` due to the limited resolution of the AM9513's 16 bit counters), and the possible resolution of the pulse width given the current parameters. The **XCMDErr** global will return a silent error 60 if the actual and requested pulse widths differed.

Example:
Fig. 11-7 shows an example of signals that would be generated by the following script (modified from the 'DAC Pulse' button in the Button Bank):

```
global DACGainTable        -- contains external gain information

put 0 into DACchannel      -- the D/A channel to be pulsed (0 or 1)
put 1000 into DACpulse     -- the analog level of the pulse (mV)
put 100513 into pulseWidth -- pulse width in µs
put -500 into DACpost      -- analog level after pulse
put 0 into TTLbit          -- toggle bit number (0-7, -1 for none)

-- adjust DAC levels w.r.t external DAC gain
get line (DACchannel+1) in DACGainTable
put round(DACpulse/it) into adjDACpulse
put round(DACpost/it) into adjDACpost

-- generate the pulse
get DACPulse (DACchannel,adjDACpulse,pulseWidth,adjDACpost,TTLbit)
put it
```

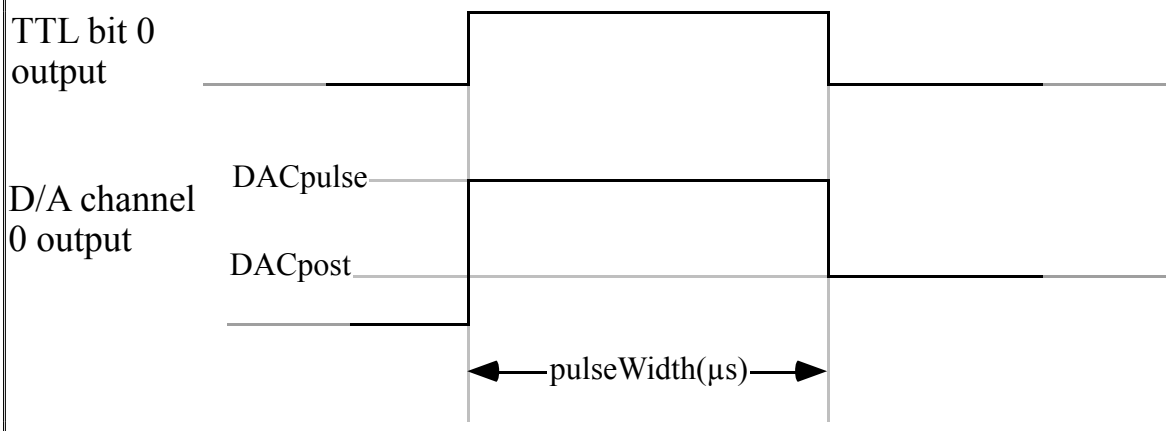


Fig. 11-7: Example of signals generated by the `DACPulse` XFCN.

Note how the parameters defining the analog levels (`DACpulse`, `DACpost`) were adjusted with respect to any external gain or attenuation (stored in `DACGainTable`). The message box will read '100514,2' indicating that the actual pulse width was 100514 instead of 100513 µs because the resolution of pulses this long is limited to 2 µs. As a percentage of the total pulse width, this limitation is insignificant. **XCMDErr** is also set to 60 to signal that the actual and requested pulse widths differed.

DesignFIRhi

Type: XFCN

Syntax:
`DesignFIRhi (fstop,fpass,dB,sampleInterval)`

Description:
Designs a Kaiser-Bessel hi-pass finite impulse response filter. The stop band extends from zero (DC) to `fstop` Hz where all components are attenuated by at least `dB` decibels (pass a positive value for `dB`). The pass band is defined as the band extending above `fpass` (Hz). `fstop` must be less than `fpass`. The sampling interval is passed in `sampleInterval` (µs, need not be integer). If `sampleInterval` is zero, you can define pass and stop bands as normalized frequencies ($0 < f < 0.5$). The steepness of the filter's roll-off is determined by how close fstop and fpass are. The closer they are however, the more coefficients will be required to implement such a filter and the more computation time each filter operation using ConvolveWave will require.

Result:
Returns an even number of filter coefficients as compressed wave as value of XFCN, always of type float. Pass the coefficient wave generated by DesignFIRhi directly to ConvolveWave to filter your signal efficiently. See ConvolveWave XFCN for more details.

Example:
You have a digitized signal sampled at 100 kHz and you want to remove low frequency components by high-pass filtering the wave at 12 kHz. The filter must roll off so that all components are attenuated to less than 100 dB below 5 kHz. Design the filter by calling `DesignFIRhi` with the appropriate parameters, then filter your signal using `ConvolveWave`:

```
global theWave,FIRCoeffs,w0
put 0 into resultType      -- return same type of wave
-- filter the wave, and return result in same global
put 5000 into fstop  -- in Hz
put 12000 into fpass
put 100 into dB
put 10 into sampleInterval  -- 100kHz is 10µs sample
interval
-- design filter coeffs
put DesignFIRhi(fstop,fpass,dB,sampleInterval) into
FIRCoeffs
-- filter it, result into global w0
put ConvolveWave ("theWave", "FIRCoeffs", resultType)
into w0
```

DesignFIRlo

Type: XFCN

Syntax:
`DesignFIRlo (fpass,fstop,dB,sampleInterval)`

Description:
Designs a Kaiser-Bessel lo-pass finite impulse response filter. The pass band extends from zero to `fpass` Hz. The stop band is defined as the band extending above `fstop` (Hz) where all components are attenuated by at least `dB` decibels (pass a positive value for `dB`). Obviously `fstop` must be greater than `fpass`. The sampling interval is passed in `sampleInterval` (µs, need not be integer). If `sampleInterval` is zero, can define pass and stop bands as normalized frequencies ($0 < f < 0.5$). The steepness of the filter's roll-off is determined by how close fpass and fstop are. The closer they are however, the more coefficients will be required to implement such a filter and the more computation time each filter operation using ConvolveWave will require.

Result:
Returns an even number of filter coefficients as compressed wave as value of XFCN, always of type float. Pass the coefficient wave generated by DesignFIRlo directly to ConvolveWave to filter your signal efficiently. See ConvolveWave XFCN for more details.

Example:
You have a digitized signal sampled at 100 kHz and you want to remove high frequency noise by low-pass filtering the wave at 12 kHz. The filter must roll off so that all components are attenuated to less than 100 dB above 35 kHz. Design the filter by calling `DesignFIRlo` with the appropriate parameters, then filter your signal using `ConvolveWave`:

```
global theWave,FIRCoeffs,w0
put 0 into resultType      -- return same type of wave
-- filter the wave, and return result in same global
put 12000 into fpass  -- in Hz
put 35000 into fstop
put 100 into dB
put 10 into sampleInterval  -- 100kHz is 10µs sample
interval
-- design filter coeffs
put DesignFIRlo(fpass,fstop,dB,sampleInterval) into
FIRCoeffs
-- filter it, result into global w0
put ConvolveWave ("theWave", "FIRCoeffs", resultType)
into w0
```

For lo-pass filtering, DesignFIRlo and ConvolveWave are the most convenient and efficient (in terms of computation speed) way to filter signal in WaveTrak.

DifferentiateWave

Type: XFCN

Syntax:
```
DifferentiateWave ("theWave",resultType)
```

Description:
Differentiates the wave in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global). The derivative is computed by taking the difference between successive points:

$$y_0 = y_1 - y_0$$
.
.
$$y_n = y_{n+1} - y_n$$
.
.
$$y_{npoints-2} = y_{npoints-1} - y_{npoints-2}$$
$$y_{npoints-1} = y_{npoints-1} - y_{npoints-2}$$

Notice that the last two points in the derivative are always equal; the one extra point is used to pad the result so that it has the same number of points as the original wave. `resultType` selects the data type of the derivative (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return the same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

Result:
Returns compressed wave as value of XFCN.

Example:
```
global theWave
put "F" into resultType
-- differentiate, making result a floating point type
put DifferentiateWave ("theWave",resultType) into w0
```

DivideWaves

Type: XFCN

Syntax:
`DivideWaves ("wave1","wave2",resultType)`

Description:
Divides each point of `wave1` by the corresponding point in `wave2`:

$$y_n = y_{n,wave1} \div y_{n,wave2}$$

If the number of points in both waves is different, stops dividing when reaches the end of shorter wave. Double quotes are included to remind you to pass the *names* of the globals containing the waves. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `wave1`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. If any element in `wave2` is zero, a divide-by-zero error (**XCMDErr** = 44) is returned and the function result is undefined.

Result:
Returns compressed wave as value of XFCN.

Example:
```
global wave1,wave2
put 0 into resultType  -- return same type as wave1
-- divide wave1 by wave2, result goes in wave1
put DivideWaves ("wave1","wave2",resultType) into wave1
```

DrawWave

Type: XCMD

Syntax:
`DrawWave gList,dispRect,topY,bottomY,baseline`

Description:
`DrawWave` is similar to `DrawWaveCoords` (see below), but does not support any cursor readout. Use this XCMD if you only want to draw several waves and not trap execution if the cursor is within `dispRect` (the display window or rectangle). The names of globals where waves are stored are passed as a comma-delimited list in `gList`. Up to 16 waves can be overlaid simultaneously; *the wave type and the number of points in all waves must be identical* or an error will result.

The screen area enclosed by `dispRect` is erased first, and the plot will be scaled to precisely fit the rectangle. For integer waves, `topY` and `bottomY` are only used to position the baseline. The wave plot is scaled so that the maximum value for that type (e.g. +2047 for a signed 12-bit wave) appears at the top of `dispRect`, and the minimum value (e.g. -2048) at the bottom of the rectangle. If topY=bottomY=0, the range defaults to the maximum integer range, and these limits do not then need to be passed explicitly; the baseline or zero level will be positioned mid-way. Float waves, however, are scaled such that values equal to `topY` and `bottomY` will be positioned at the top and bottom of `dispRect`, respectively. Values beyond this range are clipped to the region enclosed by `dispRect`.

When the **dotFlag** global is TRUE, the wave is drawn with dots only, otherwise lines connect the dots if **dotFlag** is FALSE. If `bottomY` ≤ `baseline` ≤ `topY`, a horizontal baseline is drawn for reference. The **maxPlotPoints** global determines how many points are drawn for each view.

Result:
None.

Example:

```
global w0,w1
-- std WaveTrak dispRect (left,top,right,bottom)
put "12,47,375,289" into dispRect
-- -10 to +10 V vertical range, used only to position
baseline
put 10 into topY
put -10 into bottomY
put "w0,w1" into gList  -- plot w0 & w1 together
put 0 into baseline     -- baseline @ 0 volts
DrawWave gList,dispRect,topY,bottomY,baseline
```

DrawWaveCoords

Type: XCMD

Syntax:
```
DrawWaveCoords
gList,dispRect,leftX,rightX,topY,bottomY,baseline,
Xunit,Yunit
```

Description:
`DrawWaveCoords` is a very powerful display function and is the main routine used by WaveTrak to draw your traces on the screen in real time as you navigate around the stack. The names of globals where waves are stored are passed as a comma-delimited list in `gList`. Up to 16 waves can be overlaid simultaneously; *the wave type and the number of points in all waves must be identical* or an error will result. The screen area enclosed by `dispRect` is erased first, and the plot will be scaled to precisely fit the rectangle. If the cursor is within `dispRect` when this XCMD is called, a crosshair will display the current position of the cursor in real units defined by the wave descriptors `leftX`, `rightX`, `topY`, `bottomY`, `Xunit`, `Yunit` (see the discussion on wave descriptors in the Scripting chapter). Thus, when the cursor is at the extreme top left corner of `dispRect`, the crosshair will display the values of `leftX` and `topY` as the X-Y readout. Conversely, when the cursor is at the extreme bottom right corner of `dispRect`, the readout will display the values of `rightX` and `bottomY`.

For integer waves, `topY` and `bottomY` are only used to position the baseline and calculate the correct cursor readout. The wave plot is scaled so that the maximum value for that type (e.g. +2047 for a signed 12-bit wave) appears at the top of `dispRect`, and the minimum value (e.g. -2048) at the bottom of the rectangle. If topY=bottomY=0, the range defaults to the maximum integer range, and these limits do not then need to be passed explicitly; the baseline or zero level will be positioned mid-way. Float waves, however, are scaled such that values equal to `topY` and `bottomY` will be positioned at the top and bottom of `dispRect`, respectively. Values beyond this range are clipped to the region enclosed by `dispRect`.

Pressing the option key will change the cursor into an 'expand cursor' and clicking the mouse over a segment of your wave will magnify that segment and place it at

the center of `dispRect`. You can zoom in repeatedly up to a factor of 256. Pressing the shift and option keys together and clicking the mouse will shrink the display. Double clicking the mouse will return you to

the home view from any magnification. The magnification in the X and Y directions is controlled by the globals **xMag** and **yMag**, respectively. When the **dotFlag** global is TRUE, the wave is drawn with dots only, otherwise lines connect the dots if **dotFlag** is FALSE. If `bottomY ≤ baseline ≤ topY`, a horizontal baseline is drawn for reference. The **maxPlotPoints** global determines how many points are drawn for each view.

The discussion under Trace Cards in the WaveTrak Cards chapter related to examining and zooming waves applies here since all standard WaveTrak display functions use `DrawWaveCoords`. Also, examining the existing scripts in the WaveTrak master stack will teach you more about how to use this XCMD most effectively.

Result:
None.

Example:
```
global w0,w1
-- std WaveTrak dispRect (left,top,right,bottom)
put "12,47,375,289" into dispRect
-- 0 to 1000 µs horizontal range
put 0 into leftX
put 1000 into rightX
put "µs" into Xunit
-- -10 to +10 V vertical range
put 10 into topY
put -10 into bottomY
put "V" into Yunit
put "w0,w1" into gList  -- plot w0 & w1 together
put -5 into baseline    -- baseline @ -5 volts
DrawWaveCoords
gList,dispRect,leftX,rightX,topY,bottomY,baseline,¬
Xunit,Yunit
```

FilterWaveFFThiLin

Type: XFCN

Syntax:
```
FilterWaveFFThiLin
("theWave",sampleInterval,fLo,fHi,resultType)
```

Description:
Digitally high-pass filters the wave in the global `theWave` (double quotes are included to remind you to pass the *name* of the global). *The number of points in wave must be an integral power of 2* (e.g. 512, 1024, 2048) because the FFT is used to implement the filter. The original sampling interval (in µs) is passed in `sampleInterval`. The roll-off is linear and the two knees are passed in `fLo` and `fHi` (both in Hz) as shown in Fig. 11-8:

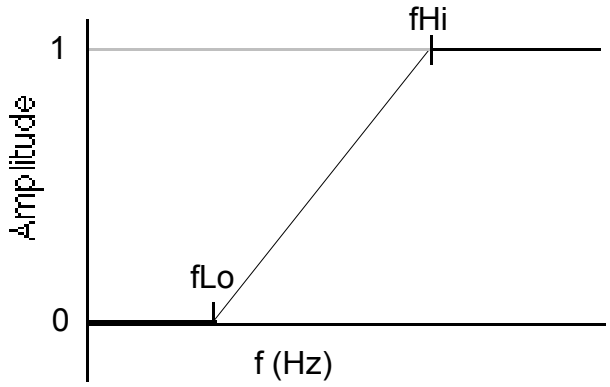


Fig. 11-8: Transfer function of digital filter implemented by `FilterWaveFFThiLin`.

`fLo` must be less than `fHi`. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Unlike log roll-off filters, linear roll-off designs completely eliminate all frequency components below `fLo`.

Technical note:

The filter is implemented by performing a forward FFT on the wave, linearly attenuating components between `fLo` and `fHi` and setting to zero all components below `fLo`, as shown in Fig. 11-8, then returning the wave to the time domain with an inverse FFT.

Result:
Returns compressed wave as value of XFCN.

Example:

```
global theWave
put 0 into resultType      -- return same type of wave
put 10 into sampleInterval -- original sampling interval in µs/point
-- filter parameters in Hz
put 1000 into fLo
put 2000 into fHi
put FilterWaveFFThiLin ("theWave",sampleInterval,fLo,fHi,¬
resultType) into theWave
```

FilterWaveFFThiLog

Type: XFCN

Syntax:
```
FilterWaveFFThiLog
("theWave",sampleInterval,f3dB,rolloff,resultType)
```

Description:
Digitally high-pass filters the wave in the global `theWave` (double quotes are included to remind you to pass the *name* of the global). *The number of points in wave must be an integral power of 2* (e.g. 512, 1024, 2048) because the FFT is used to implement the filter. The original sampling interval (in µs) is passed in `sampleInterval`. The roll-off is logarithmic (`rolloff`, in dB/octave). Roll-offs are usually passed as positive reals to produce attenuation below the -3 dB frequency; negative roll-off values are accepted and will result in emphasis of those frequencies instead. Regardless of the sign of the roll-off parameter, the DC component is always returned as zero (otherwise it would be infinite with a negative roll-off value). The -3 dB frequency is passed in `f3dB` (in Hz, must be > zero) as shown in Fig. 11-9:

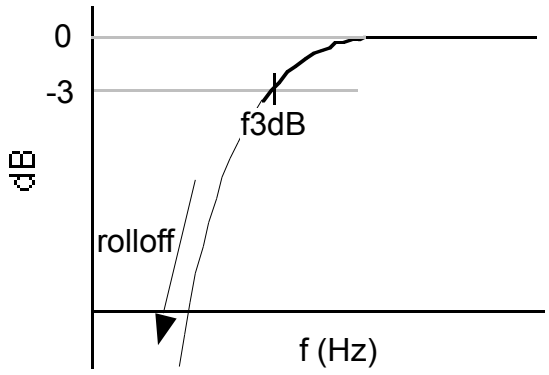


Fig. 11-9: Transfer function of digital filter implemented by `FilterWaveFFThiLog`.

`resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

Technical note:

The filter is implemented by performing a forward FFT on the wave, logarithmically attenuating components below `f3dB` at a rate of `rolloff` dB per octave as shown in Fig. 11-9, then returning the wave to the time domain with an inverse FFT.

Result:
Returns compressed wave as value of XFCN.

Example:

```
global theWave
put 0 into resultType      -- return same type of wave
put 10 into sampleInterval -- original sampling interval in µs/point
put 1000 into f3dB         -- 3dB frequency in Hz
put 24 into rolloff        -- in dB/octave
put FilterWaveFFThiLog ("theWave",sampleInterval,f3dB,rolloff,¬
resultType) into theWave
```

FilterWaveFFTloLin

Type: XFCN

Syntax:
```
FilterWaveFFTloLin
("theWave",sampleInterval,fLo,fHi,resultType)
```

Description:
Digitally low-pass filters the wave in the global `theWave` (double quotes are included to remind you to pass the *name* of the global). *The number of points in wave must be an integral power of 2* (e.g. 512, 1024, 2048) because the FFT is used to implement the filter. The original sampling interval (in µs) is passed in `sampleInterval`. The roll-off is linear and the two knees are passed in `fLo` and `fHi` (both in Hz) as shown in Fig. 11-10:

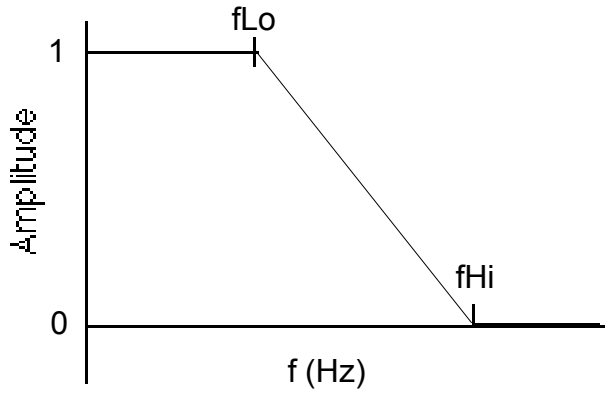


Fig. 11-10: transfer function of digital filter implemented by `FilterWaveFFTloLin`.

`fLo` must be less than `fHi`. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Unlike log roll-off filters, linear roll-off designs completely eliminate all frequency components above `fHi`.

Technical note:

The filter is implemented by performing a forward FFT on the wave, linearly attenuating components between `fLo` and `fHi`, and setting to zero all components above `fHi`, as shown in Fig. 11-10, then returning the wave to the time domain with an inverse FFT.

Result:
Returns compressed wave as value of XFCN.

Example:

```
global theWave
put 0 into resultType      -- return same type of wave
put 10 into sampleInterval -- original sampling interval in µs/point
-- filter parameters in Hz
put 1000 into fLo
put 2000 into fHi
put FilterWaveFFTloLin ("theWave",sampleInterval,fLo,fHi,¬
resultType) into theWave
```

FilterWaveFFTloLog

Type: XFCN

Syntax:
```
FilterWaveFFTloLog
("theWave",sampleInterval,f3dB,rolloff,resultType)
```

Description:
Digitally low-pass filters the wave in the global `theWave` (double quotes are included to remind you to pass the *name* of the global). *The number of points in wave must be an integral power of 2* (e.g. 512, 1024, 2048) because the FFT is used to implement the filter. The original sampling interval (in µs) is passed in `sampleInterval`. The roll-off is logarithmic (`rolloff`, in dB/octave). Roll-offs are usually passed as positive reals to produce attenuation above the -3 dB frequency; negative roll-off values are accepted and will result in emphasis of those frequencies instead. Emphasis of higher frequencies can be used to sharpen edges, but tends to result in noisy signals. The -3 dB frequency is passed in `f3dB` (in Hz, must be > zero) as shown in Fig. 11-11:

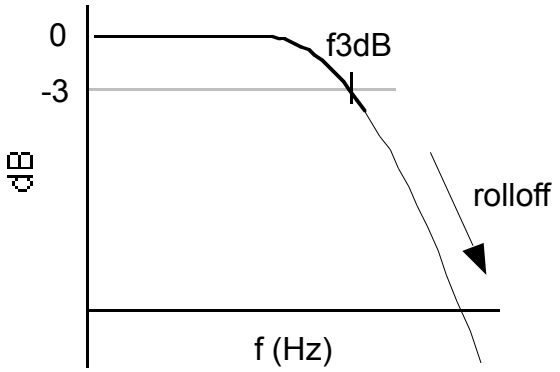


Fig. 11-11: transfer function of digital filter implemented by `FilterWaveFFTloLog`.

`resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

Technical note:

The filter is implemented by performing a forward FFT on the wave, logarithmically attenuating components above `f3dB` at a rate of `rolloff` dB per octave as shown in Fig. 11-11, then returning the wave to the time domain with an inverse FFT.

Result:
Returns compressed wave as value of XFCN.

Example:

```
global theWave
put 0 into resultType      -- return same type of wave
put 10 into sampleInterval -- original sampling interval in µs/point
put 1000 into f3dB         -- 3dB frequency in Hz
put 24 into rolloff        -- in dB/octave
put FilterWaveFFTloLog ("theWave",sampleInterval,f3dB,rolloff,¬
resultType) into theWave
```

FilterWaveFIR

Type: XFCN

Syntax:
`FilterWaveFIR ("theWave","FIRCoeffs",resultType)`

Description:
Implements a finite impulse response (non-recursive) digital filter. The wave to be filtered is passed in global `theWave`, and the coefficients describing the filter are passed in global `FIRCoeffs` (double quotes are included to remind you to pass the *names* of the globals). Unlike the FFT-based filters above, the number of points in the wave need not be an integral power of two. You must design the filter beforehand using another application such as Igor Filter Design Lab ('IFDL', WaveMetrics, Lake Oswego, OR), then import the coefficients into WaveTrak by pasting them into the coefficients field on the Digital Filter Parameters card (see the chapter on WaveTrak cards for more details). When you paste the coefficients using the button, the ASCII values are saved in the field, and are also converted into a WaveTrak wave, of type float, and copied into the global `FIRCoeffs`.

> Technical note:
>
> The conversion is done for speed so that the ASCII table of filter coefficients doesn't have to be converted to an array of floats each time the filter routine is invoked -- converting compressed WaveTrak waves is much quicker.

`resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

The advantage of using FIR filtering is greater speed than FFT (for a small number of coefficients, e.g. <100), and the ability to design any number of filter types such as lo-, hi-, band-pass and notch filters. Consult the Igor Filter Design Lab manual for more details. The disadvantage is that the filter needs to be designed beforehand and cannot be changed on-the-fly.

Result:
Returns filtered, compressed wave as the value of the XFCN. The filtered wave is shifted left by n/2 samples (where n is the number of filter coefficients) so that the original and filtered waves are in phase (FIR filtering shifts the output right by n/2 samples). Also, the first and last n/2 points are returned unfiltered, because there are n/2 points missing before the start of the wave, and after the end of the wave, required to compute the output.

Example:
You have a digitized signal sampled at 100 kHz and you want to remove high frequency noise by low-pass filtering the wave at 10 kHz. You must first design the filter: start IFDL and select "Initialize IFDL Params..." from the Macro menu. The sampling rate is left at the normalized default of 1 Hz. You decide that a McClellan-Parks-Rabiner lo-pass filter is required, therefore select "FIR Filter Designs" from the Macro menu and choose MPR Low Pass. The end of the normalized passband frequency will be 0.1 (10 kHz/100 kHz), with a maximum error of no more than 0.5 dB, and the stopband begins at 0.2 (20 kHz) with a minimum attenuation of 60 dB. Click 'Continue' and the filter characteristics are graphically displayed; a filter with 22 coefficients is computed that will satisfy your requirements. To transfer the filter to WaveTrak you then copy the coefficients from the 'CoefsTable' window and paste into the Digital Filters card by clicking on the 'Paste Coefs' button. This copies the coefficients into the field and converts the ASCII values into a WaveTrak wave in the global `FIRCoeffs`. You can now implement your new filter using the `FilterWaveFIR` XFCN:

```
global theWave,FIRCoeffs
put 0 into resultType      -- return same type of wave
-- filter the wave, and return result in same global
get FilterWaveFIR ("theWave","FIRCoeffs",resultType)
put it into theWave
```

Go to one of the three sample traces in the first root of WaveTrak and press the 'FIR filter' button to see how the existing filter affects the signal.

GetScrap

Type: XFCN

Syntax:
```
GetScrap()
```

Description:
Returns the contents of the clipboard. If clipboard does not contain text, an error is returned (**XCMDErr** = 2). The clipboard is also called the *scrap* in Macintosh parlance, hence the name of this function. Use this XFCN to import text data into HyperCard variables.

Result:
Returns contents of clipboard as value of XFCN.

Example:
```
put GetScrap() into x
```

Note that even though `GetScrap` has no parameters, the parentheses must be included.

GetWaveStats

Type: XFCN

Syntax:
```
GetWaveStats ("theWave",elementNum)
```

Description:
Computes various statistics on the wave stored in global `theWave` (double quotes are included to remind you to pass the wave by name, not by value). The result of the function is returned as a table consisting of 12 lines separated by carriage returns:

Line 1: number of points in the wave.
Line 2: wave type (e.g. -12: signed 12 bit integer; F: single precision floating point.
Line 3: raw mean of all points in the wave.
Line 4: raw root-mean-square (RMS) value of all points.
Line 5: standard deviation (= RMS with DC removed) of all points.
Line 6: smallest possible value for this wave type (e.g. -2048 for type = -12, 0 for type = 16, -3.2E+38 for type = F).
Line 7: largest possible value for this wave type (e.g. 2047 for type = -12, 65535 for type = 16, 3.2E+38 for type = F)
Line 8: minimum actual value (without scaling) in the form: element number (0 to npoints-1), value.
Line 9: maximum actual value (without scaling) in the form: element number (0 to npoints-1), value.
Line 10: range (line 9 - line 8).
Line 11: value of point number `elementNum` ('NaN' if `elementNum` parameter omitted).
Line 12: size of encoded wave in bytes, including terminal null.

The `elementNum` parameter is optional. If it is omitted, line 11 returns 'NaN' (a code for 'not a number').

Result:
Returns a table of wave statistics as the value of the XFCN.

Example 1:
This is what `GetWaveStats` will return if you pass it the sine wave in the first sample trace card:

```
global theWave
get GetWaveStats ("theWave",100)
put it into theStats
```

The variable `theStats` will contain (without comments of course):

```
2048      the wave has 2048 points.
-12       the wave is a signed 12 bit integer type.
41.4      the mean value.
729.4     the RMS value.
728.3     the standard deviation.
-2048     the minimum legal value for a signed 12 bit integer type.
2047      the maximum legal value for a signed 12 bit integer type.
315,-996  the smallest actual value was -996 at point no. 315.
111,1099  the largest actual value was 1099 at point no. 111.
2095      the range was 2095 (1099-(-996)).
1069      the value of point no. 100 was 1069.
2053      the wave takes up 2053 bytes when compressed and encoded.
```

The real values (mean, RMS and standard deviation) are converted according to the **XYCoordYformat** global. You have to adjust the results in accordance with the wave's actual full scale range (±10 volts in this example). See the following example.

Example 2:

Given the example above, you want to compute the true time and voltage values for the maximum point in the wave. From line 9 in `theStats`, the raw values for the maximum point are `111,1099`. Get the actual *time* by multiplying the point number by the sampling interval = 111*25 = 2775 µs (the sampling interval is stored in line 3 of the 'HParams' field in every trace card). Getting the voltage is easy if you use the `translateToReal` handler in the stack script (see the chapter on WaveTrak handlers for details):

```
global topY,bottomY  -- these globals initialized at
openCard
put line 2 in theStats into theType  -- the data type,
-12
get translateToReal (1099,bottomY,topY,theType)
```

`it` will now contain the true voltage value of 5369.96337 mV (`Yunit` tells us that the original unit was mV). Therefore, the highest point in the wave occurred at 2775 µs and attained a level of 5.369 volts.

InitWaveK

Type: XFCN

Syntax:
```
InitWaveK (npoints,K,resultType)
```

Description:
Creates a new wave consisting of `npoints` elements, and initializes each point to a constant, `K`. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 in this context will generate an error. If an integer wave is requested by `resultType` and `K` is out of range, a silent overflow error (**XCMDErr** = 42) is returned and the wave is initialized to the maximum or minimum allowable value. If an integer wave is requested and `K` is fractional, it is rounded first.

Result:
Returns compressed wave as value of XFCN.

Example:
```
put 2048 into npoints
put 100.7 into K
put -12 into resultType
put InitWaveK (npoints,K,resultType) into w0
```

`w0` will be a signed, 12 bit integer wave consisting of 2048 points, all equal to 101.

InitWaveNoise

Type: XFCN

Syntax:
```
InitWaveNoise (npoints,pkAmpl,offset,resultType)
```

Description:
Creates a new wave consisting of `npoints` elements, and initializes it to random white noise. The *peak* amplitude of the noise is defined by `pkAmpl` (therefore the range will be twice `pkAmpl`), and the mean DC level by `offset`. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 in this context will generate an error. If an integer wave is requested by `resultType` and some points are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range points are set to the maximum or minimum allowable value.

Result:
Returns compressed wave as value of XFCN.

Example:
```
put 2048 into npoints
put 1000 into pkAmpl
put -1000 into offset
put -12 into resultType
put InitWaveNoise (npoints,pkAmpl,offset,resultType)
into w0
```

`w0` will be a signed, 12 bit integer wave consisting of 2048 points initialized to random white noise, with values ranging from -2000 to 0.

InitWaveSin

Type: XFCN

Syntax:
```
InitWaveSin
(npoints,cycles,pkAmpl,phase,offset,resultType)
```

Description:
Creates a new wave consisting of `npoints` elements, and initializes it to a sine wave. The *peak* amplitude of the sine wave is defined by `pkAmpl` (therefore the range will be twice `pkAmpl`), and the mean DC level by `offset`. `cycles` defines the number of cycles in the wave (need not be an integer), and the phase is defined by `phase` in degrees. Mathematically:

$$y_n = \texttt{pkAmpl} * \sin(f * n + \theta) + \texttt{offset}$$

where:

n ranges from 0 to `npoints`-1

$f = 2\pi * \texttt{cycles}/\texttt{npoints}$

$\theta = 2\pi * \texttt{phase}/360$ (phase converted to radians)

`resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 in this context will generate an error. If an integer wave is requested by `resultType` and some points are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range points are set to the maximum or minimum allowable value.

Result:
Returns compressed wave as value of XFCN.

Example:
Generate 3 cycles of a cosine wave.

```
put 2048 into npoints
put 3 into cycles
put 1 into pkAmpl
put 90 into phase
put 1 into offset
put "F" into resultType
put InitWaveSin
(npoints,cycles,pkAmpl,phase,offset,resultType)¬
into w0
```

`w0` will be a floating point wave consisting of 2048 points initialized to 3 cycles of a cosine, with values ranging from 0 to 2. Note that the `cycles`, `phase`, `pkAmpl` and `offset` parameters give you great flexibility in creating the type of trigonometric function you need. Combining this function with other math functions lets you create virtually any trig function (e.g. use this XFCN to create a sine and cosine, then use `DivideWaves` to compute a tan function).

IntegrateWave

Type: XFCN

Syntax:
`IntegrateWave ("theWave",baseline,normalize,resultType)`

Description:
Integrates the wave in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global). The integral is a cumulative sum of all points. Mathematically:

$$y_n = \sum_{i=0}^{n} w_i - \text{baseline}$$

where:

n ranges from 0 to `npoints`-1

y_n are points in the result

w_i are points in the `theWave`

Integration usually produces very large positive or negative values that usually have to be scaled down before being overlaid or otherwise compared with the original wave. If you set `normalize` to TRUE, the result will be normalized so that the greatest value will be +1 (or the largest negative value will be -1). This way you will always know the range of the integral and can easily scale it with `MultWaveK`; otherwise you would need to perform several extra steps to find out the minimum and maximum values with `GetWaveStats`. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Since normalized waves range only from -1 to +1, they are always returned as float types.

Result:

Returns compressed wave as value of XFCN.

Example:

```
put TRUE into normalize     -- normalize integrated
wave to ±1 range
global w0,theWave           -- wave copied to theWave
at openCard
global topY,bottomY,baseline  -- defined at openCard
-- convert baseline from real units to binary
put -12 into resultType     -- wave is a 12 bit signed
integer type
put translateToBinary(baseline,bottomY,topY,resultType)
into binBase

-- integrate
put IntegrateWave
("theWave",binBase,normalize,resultType) into w0

-- now scale integrated wave to roughly match size of
original
put MultWaveK("w0",2000,resultType) into w0  -- return
integer wave
```

Note that the trace baseline is stored in real units (e.g. mV) in the trace cards and must be converted to its binary equivalent before integrating, using the `translateToBinary` handler in the stack script. Because `w0` is a normalized integral, we can scale it using `MultWaveK`, knowing that the result will be within a range of ±2000. `MultWaveK` will also convert the new scaled result to a 12 bit integer type so the original and the integral can be overlaid with `DrawWaveCoords`. This strategy is used in the 'Integrate' button in the Button Bank.

MeanWave

Type: XFCN

Syntax:
`MeanWave ("theWave",sampleInterval,startTime,endTime)`

Description:
Computes the raw mean of points between `startTime` and `endTime` µs *included*, in wave stored in global `theWave` (double quotes are included to remind you to pass the *name* of the global). `sampleInterval` is the original sampling interval in µs, `startTime` and `endTime` define the segment of the wave to be averaged. Passing -1 in `endTime` tells the XFCN to continue to the last point. For example, pass `startTime` = 0,`endTime` = -1 to compute mean of entire wave, or `startTime`,`endTime` = -1 to compute mean from `startTime` µs to end of wave. Use for computing baselines from a segment of a wave, or the overall mean DC level of a signal.

Result:
Returns a real value formatted according to **XYCoordYformat** global. This is a raw mean i.e. a sum of integer or floating point values, which must be corrected for full-scale range, A/D coding, amplifier gain, etc...

Example:
Acquire a wave from A/D channel 0 while delivering a 100 µs stimulus pulse from timer channel 1. You know that the stimulus artifact lasts 500 µs, so you want to compute the mean value of the response (in real mV) starting at 500 µs to the end, to avoid the contaminating artifact. The number of points and the sample interval are defined elsewhere as globals:

```
global sampleInterval,npoints,FSTable,theWave,ADCbits

put 0 into startMUX      -- the A/D channel
put startMUX into endMUX
put 100 into pulseWidth  -- width of stimulus pulse
put 0 into preTrig
put 1 into timerChannel

-- stimulate and acquire the signal
get AcqWaveTimer
(sampleInterval,npoints,startMUX,endMUX,¬
timerChannel,preTrig,pulseWidth,"theWave")

-- avoid artifact, compute mean from 500 µs to end
put MeanWave ("theWave",sampleInterval,500,-1) into
theMean

-- translate into mV
put item 1 in line (startMUX+1) in FSTable into
minFullScale
put item 2 in line (startMUX+1) in FSTable into
maxFullScale
get
translateToReal(theMean,minFullScale,maxFullScale,ADCbi
ts)
```

The raw computed mean in `theMean` had to be translated from a mean of binary samples into real mV using the `translateToReal` handler in the stack script.

MultWaveK

Type: XFCN

Syntax:
`MultWaveK ("theWave",K,resultType)`

Description:
Multiplies every point in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global) by a constant `K` (need not be an integer). `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Useful for scaling waves for overlays, or numerically implementing gain or attenuation. Pass -1 in `K` if you want to invert a wave. Pass 1/`K` if you want to divide each point by a constant `K`.

Result:
Returns compressed wave as value of XFCN.

Example:
```
global theWave
put 100 into K
put 0 into resultType
-- multiply every point in theWave by 100, return same
data type
put MultWaveK ("theWave",K,resultType) into theWave
-- divide every point in theWave by 50, return float
data type
put 50 into K
put "F" into resultType
put MultWaveK ("theWave",1/K,resultType) into theWave
```

MultWaves

Type: XFCN

Syntax:
```
MultWaves ("wave1","wave2",resultType)
```

Description:
Multiplies two waves together, point by point:

$$y_n = y_{n,wave1} * y_{n,wave2}$$

If the number of points in both waves is different, stops multiplying when reaches the end of shorter wave. Double quotes are included to remind you to pass the *names* of the globals containing the waves. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `wave1`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Useful for multiplying a signal by an envelope, as in amplitude modulation (AM), or for windowing a wave prior to FFT (the `Window` XFCN implements several standard window functions).

Result:
Returns compressed wave as value of XFCN.

Example:
```
global wave1,wave2
put 0 into resultType
-- multiply wave1 by wave2, result goes in wave1
put MultWaves ("wave1","wave2",resultType) into wave1
```

OverlayWave

Type: XCMD

Syntax:
`OverlayWave gList,dispRect,topY,bottomY,baseline`

Description:
`OverlayWave` is similar to `DrawWave`, except that it does not erase what was already drawn in the `dispRect`. Call `DrawWave` for the first set of waves, to erase the `dispRect`, then call `OverlayWave` to overlay additional waves over those previously drawn. This XCMD is very useful if you want to draw several waves simultaneously but do not have access to all the data at once. Cursor readout and zooming is not supported by `OverlayWave`. The only way to draw multiple waves and zoom them simultaneously is to pass all of them in a single call to `DrawWaveCoords`.
See `DrawWave` for details about the parameter list.

Result:
None.

Example:
```
global w0,w1,w2,w3
-- std WaveTrak dispRect (left,top,right,bottom)
put "12,47,375,289" into dispRect
-- -10 to +10 V vertical range, used only to position
baseline
put 10 into topY
put -10 into bottomY
put "w0,w1" into gList  -- plot w0 & w1 together
put 0 into baseline  -- baseline @ 0 volts
DrawWave gList,dispRect,topY,bottomY,baseline
.
.
put "w2,w3" into gList
-- overlay w2 & w3 without erasing w0 & w1
OverlayWave gList,dispRect,topY,bottomY,baseline
```

PowerSpectrum

Type: XFCN

Syntax:
```
PowerSpectrum ("theWave",dB,floor)
```

Description:
Computes frequency (power) spectrum of wave in global `theWave` (double quotes are included to remind you to pass the *name* of the global containing the wave):

$$y_n = Re_n^2 + Im_n^2$$

where:

y_n = nth frequency component
Re_n = real part of nth element after FFT
Im_n = imaginary part of nth element after FFT

The number of points in `theWave` must be an integral power of 2 for the FFT. If `db` = TRUE, converts spectrum to a log scale and returns elements in dB normalized to maximum value (= 0 dB). Values < `floor` will be clipped to `floor`; this is to avoid very large negative components with a log scale. If `floor` = 0, small values are not clipped and 0 elements in spectrum (which should be -∞ on a log scale) are returned as -3.403E+38 (the smallest single precision floating point number, because HyperCard does not recognize the -INF symbol). dB values are computed as follows:

$$dB = 10 \log \left(\frac{y_n}{y_{max}} \right)$$

Result:
Returns compressed wave as value of XFCN. Result is always a floating point wave (type "F").

Example:

```
global w0,theWave
put TRUE into dB   -- display on a log scale,
normalized to 0 dB
put -80 into floor -- clip very small components to -80
dB
put PowerSpectrum ("theWave",dB,floor) into w0
```

PutScrap

Type: XCMD

Syntax:
```
PutScrap x
```

Description:
Copies contents of variable `x` (global or local) to the clipboard. Note that double quotes should not enclose the variable name since it is passed by value, not by reference. Unlike `GetScrap`, this is an XCMD so no parentheses should be used. The clipboard is also called the *scrap* in Macintosh parlance, hence the name of this command. Use this XFCN to export text data from WaveTrak.

Result:
None.

Example:
```
put "1,2,3,4" into x
PutScrap x
```

The clipboard will contain the string `1,2,3,4`.

PutToGlobal

Type: XCMD

Syntax:
```
PutToGlobal value,"destgName"
```

Description:
Copies contents of variable into global `destgName` (double quotes are included to remind you to pass the *name* of the global). This is a very useful command whose importance may not be immediately obvious. WaveTrak frequently uses arrays of waves, which can be passed using the names of the global variables in a comma-delimited list. The `AcqWave` commands are a good example. You can extract each wave by using the built-in HyperCard function `the value of (item n in gList)`, for example (the Scripting chapter discusses this technique in more detail). The `AcqWave` command does the job of filling the globals with data. But what if you want to create your own array of waves and have to fill each global with data yourself? This problem comes up in operations such as the 'Overlay Waves...' menu item under the 'Analysis' menu in trace cards (see the `TrOverlay` handler in the trace background script). We have a list of global names, and we have to iterate and fill each corresponding global variable with data. In essence, we need a function that does the opposite of the standard `the value of ()` HyperCard function. The example below illustrates this point.

Result:
None.

Example:
You need to write a script that will generate *up to* 16 sine waves, place them in globals, and compile the names of the globals into a `gList` so that a command like `DrawWaveCoords` can draw them together. This would be easy if the number of sine waves was constant. Instead, the number of waves is determined by a variable (≤ 16), and you don't know beforehand how many sines you must generate. You therefore need to do this in a loop:

```
-- up to 16 globals in wave array to receive sines
global
w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15
global gList -- std global containing list of global
names

-- put the global names in a list for easy access
put
"w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15"
into¬
gNameList

-- how many sines to generate?
ask "Enter number of sine waves:" with 1
put it into nWaves

-- define the sine wave
put 2048 into npoints
put 2000 into pkAmpl
put 0 into phase
put 0 into offset
put -12 into resultType

put empty into gList  -- clear old names
repeat with waveCtr=1 to nWaves
 put waveCtr into cycles  -- change number of cycles
for each sine
 -- make a new sine wave
 get
InitWaveSin(npoints,cycles,pkAmpl,phase,offset,resultTy
pe)
```

```
 -- put NAME of next global from gNameList into local
var gName
 put item waveCtr in gNameList into gName
  -- copy sine wave to next global in list
 PutToGlobal it,gName
```

```
 -- append next global name for plotting, etc.
 put gName into item waveCtr in gList
end repeat
```

You define the 16 globals, then put their names into a comma-delimited list (`gNameList`). Each iteration of the loop generates a new sine wave, and `PutToGlobal` is used to copy the new wave to the next global, determined by the names in `gNameList`. `gList` is the standard global containing a list of global names containing valid data in a trace card (see the Scripting chapter). This ensures that your sine waves will work as expected in every trace card (if you put the example script into a button on a trace card and execute it, your sine waves will be drawn and zoomed normally).

`PutToGlobal` is useful when you only have access to the *name* of a global, and the names change as in successive iterations of a loop, as shown in the example. Passing a global name as a constant in a quoted string (e.g. `PutToGlobal it,"w0"`) makes little sense because `put it into w0` would do just as well.

ReadTTLbit

Type: XFCN

Syntax:
`ReadTTLbit (bitNumber)`

Description:
Reads the current value of bit `bitNumber` (0 to 7) of the TTL input port.

> Technical note:
>
> Keep in mind that HyperTalk scripts execute relatively slowly compared to externals written in C or assembly. Therefore you shouldn't use this XFCN for polling the TTL port to trigger some action like an acquisition when the bit changes state. Use `AcqWaveOnTTL` instead.

Result:
Returns 0 if bit is TTL low, or 1 if bit is TTL high.

Example:

```
ask "Which bit number do you want to read (0-7)?" with 0
put it into bitNumber
-- read the bit
get ReadTTLbit (bitNumber)
put "State of TTL input bit " & bitNumber &" = " & it
```

ReadTTLbyte

Type: XFCN

Syntax:
`ReadTTLbyte (inOut)`

Description:
Reads the current byte (8 bits) at either the TTL input port (`inOut` = 0) or the TTL output port read back register (`inOut` = 1). The latter is useful for monitoring the current output at the digital output port.

> Technical note:
>
> Keep in mind that HyperTalk scripts execute relatively slowly compared to externals written in C or assembly. Therefore you shouldn't use this XFCN for polling the TTL port to trigger some action like an acquisition when the bits change state. Use `AcqWaveOnTTL` instead.

Result:
Returns unsigned byte value of input or output port (0 to 255).

Example:

```
-- 0=read input port, 1=read  contents of output port
get ReadTTLbyte (0)  -- read the input port
put "TTL input port value: " & it
```

ScrapToComma

Type: XFCN

Syntax:
`ScrapToComma()`

Description:
This XFCN converts the contents of the clipboard from tab-delimited to comma-delimited format. Also converts commas to slashes ('/') to avoid extra unwanted columns when HyperCard attempts to interpret the table. A terminal null (ASCII zero) is appended if none exists, for HyperCard string compatibility. The clipboard is also called the *scrap* in Macintosh parlance, hence the name of this function.

Use this XFCN to convert standard lists and tables, which are tab-delimited, to HyperCard format (comma-delimited) for importing from spreadsheets, etc.

Result:
The contents of the clipboard are converted in place. Returns as the value of the function the number of tabs converted to commas.

Example:
The clipboard contains the following tab-delimited table (3 columns, 2 rows). Commas do not separate columns here:

```
     1     2     3,last item
     4,5,6 5     6

-- convert
put ScrapToComma()
```

The contents of the clipboard will now be:

```
     1,2,3/last item
     4/5/6,5,6
```

Each tab character (which separated columns) has been replaced by a comma. The message box reads 4 indicating that 4 tabs were converted to commas. HyperCard can now separate each of the three *items* separated by commas (e.g. '`3/last item`' is one item as far as HyperCard is concerned); the original commas were converted to slashes to retain the original number of columns.

ScrapToTab

Type: XFCN

Syntax:
```
ScrapToTab()
```

Description:
This XFCN converts the contents of the clipboard from comma-delimited to tab-delimited format. Also adds a terminal null (ASCII zero) if none exists, for HyperCard string compatibility. Existing tabs are unchanged. The clipboard is also called the *scrap* in Macintosh parlance, hence the name of this function. Use this XFCN to convert HyperCard lists and tables, which are comma-delimited, to standard Mac export format (tab-delimited) for exporting to spreadsheets, etc.

Result:
The contents of the clipboard are converted in place. Returns as the value of the function the number of commas converted to tabs.

Example:
The clipboard contains the following (3 HyperCard items per line):

```
  1,2,3/last item
  4,5,6

-- convert
put ScrapToTab()
```

The contents of the clipboard will now be:

```
  1    2    3/last item
  4    5    6
```

The message box reads 4 indicating that 4 commas were converted to tabs.

SetADGain

Type: XCMD

Syntax:
```
SetADGain theGain
```

Description:
This XCMD sets the gain of the on-board programmable gain amplifier on the MacADIOS II card. `theGain` must be 1, 10 or 100.

> Technical note:
>
> The appropriate bit pattern is written to bits 0 and 1 of the MacADIOS II Mode register; bits 2-7 are unconditionally set to 0.

Result:
None.

Example:

```
SetADGain 10
```

The gain of the programmable gain amplifier is set to 10.

StartPulseTrain

Type: XFCN

Syntax:
`StartPulseTrain (timerChannel,pulseWidth,period)`

Description:
Generates a continuous train of TTL pulses at the AM9513 timer output `timerChannel` (1 to 5). The pulse train will continue after the function returns, until `StopPulseTrain` is called, or until another function uses that channel for another operation. The pulse train will consist of a TTL high lasting `pulseWidth` μs, with a repetition rate of `period` μs (Fig. 11-12). The maximum `pulseWidth` and/or `period` = 131000000 (= 131 seconds). `pulseWidth` must be less than `period`, and both must be positive integers.

Counters `timerChannel` and `timerChannel`-1 are used (if `timerChannel` = 1, counter 5 is used [**Important note**: counter 5 is used by *all* acquisition functions, therefore a pulse train at counter 1 will be aborted by any subsequent analog acquisition such as `AcqWave` or `AcqWaveDAC` for example]). The output of counter `timerChannel`-1 will be TTL low for the duration of the train. Make sure that other operations are not in progress on the timer channels affected by this function. Because the AM9513 has 16 bit counters, the resolution of short pulses at long repetition rates may be limited (see example). If the true pulse width would have differed by more than 30% from what was requested in the parameter list, `StartPulseTrain` returns with an **XCMDErr** = 76 and does not generate the pulse train. If the true pulse width differed by less than 30%, the train is generated and a silent error 60 is returned in **XCMDErr**.

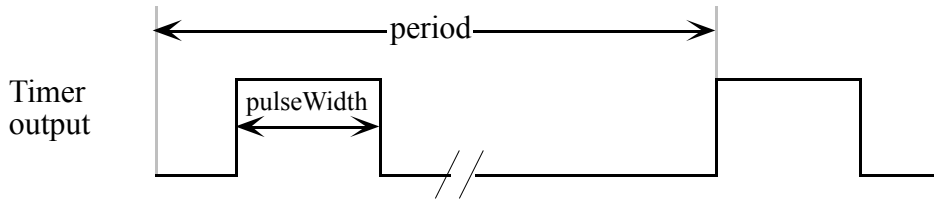


Fig. 11-12: Pulse train generated by `StartPulseTrain` XFCN.

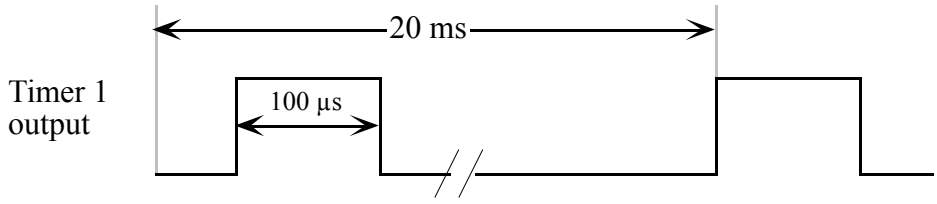
Result:
The value of the XFCN returns a comma-delimited list containing the actual width of the pulse (which may differ

from `pulseWidth` due to the limited resolution of the AM9513's 16 bit counters), the actual period, and the possible resolution of the pulse width given the current parameters.

Example 1:
Generate 100 µs pulses at 50 Hz from timer output 1:

```
put 1 into timerChannel        -- the timer channel
put 100 into pulseWidth        -- pulse width in µs
put (1/50)*1000000 into period -- period in µs
-- start the pulse train
get StartPulseTrain (timerChannel,pulseWidth,period)
put it
```

The signal at the output of timer 1 would be as shown:

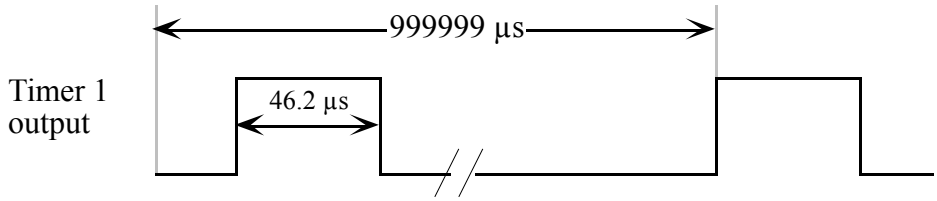


The message box will read '100,20000,1' indicating that the actual pulse width and period were exactly as requested (**XCMDErr** = 0), and the resolution of these parameters is 1 µs. The output of timer 5 will be TTL low, even though timer 5 is used internally to clock timer 1. Remember that timer 5 is used by the MacADIOS II card to clock the A/D converter, so if you subsequently call any of the `AcqWave` commands, the pulse train will be aborted. It is therefore a good idea to only use timers 2 to 4 for continuous pulse train generation.

Example 2:
Generate 47 µs pulses at 1 second intervals from timer output 1:

```
put 1 into timerChannel   -- the timer channel
put 47 into pulseWidth    -- pulse width in µs
put 1000000 into period   -- period in µs
-- start the pulse train
get StartPulseTrain (timerChannel,pulseWidth,period)
put it
```

The signal at the output of timer 1 would be as shown:



The message box will read '46,999999,15'. Note that the true pulse width is 46.2 µs, but the reported value is rounded to the nearest microsecond. **XCMDErr** is set to 60 to indicate that the actual and requested widths are different. The true period is close at 999999 µs. The possible resolution of these parameters is reported as 15 µs, but neither 46 nor 999999 are multiples of 15. The resolution was actually 15.4 µs but it too was rounded, so that the true pulse width was generated more accurately than what would have been possible with a resolution of 15 µs (i.e. 45 µs pulse width).

Example 3:
Similar to example 2, generate 47 µs pulses at 2 second intervals from timer output 1:

```
put 1 into timerChannel   -- the timer channel
put 47 into pulseWidth    -- pulse width in µs
put 2000000 into period   -- period in µs
-- start the pulse train
get StartPulseTrain (timerChannel,pulseWidth,period)
put it
```

This time, no pulse train is generated, and an **XCMDErr** = 76 is returned instead. The message box will read '61,2000016,31' indicating that the best you can do is a pulse 61 µs wide. This differed by more than 30% from the 47 µs you requested, so the signal was not generated. You have to adjust the pulse width and/or period. The possible resolution with such a long period is now 31 µs. In general, *the greater the difference between* `pulseWidth` *and* `period`*, the greater the chance that* `StartPulseTrain` *will not be able to generate a pulse with the precision requested.*

Technical note:

Due to the design of the AM9513 chip, it is not possible to concatenate two counters internally so that one edge-triggers another. External connections between the output of one channel and the gate of another channel are required.

StopPulseTrain

Type: XCMD

Syntax:
`StopPulseTrain timerChannel`

Description:
Stops the pulse train being generated at the AM9513 timer output `timerChannel` (1 to 5). Use this XCMD to stop the pulse train generated by `StartPulseTrain`. Counters `timerChannel` and `timerChannel`-1 are disarmed (see `StartPulseTrain`). Make sure that other operations are not in progress on the timer channels affected by this command.

Result:
None.

Example 1:
Generate 100 µs pulses at 1 ms intervals from timer output 1:

```
put 1 into timerChannel          -- the timer channel
-- start the pulse train
get StartPulseTrain (timerChannel,100,1000)
-- stop pulse train
StopPulseTrain timerChannel      -- timer 5 is also
disarmed
```

SubtractWaves

Type: XFCN

Syntax:
`SubtractWaves ("wave1","wave2",resultType)`

Description:
Subtract `wave2` from `wave1`, point by point:

$$y_n = y_{n,wave1} - y_{n,wave2}$$

If the number of points in both waves is different, stops subtracting when reaches the end of the shorter wave. Double quotes are included to remind you to pass the *names* of the globals containing the waves. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `wave1`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value. Useful for subtracting some underlying contaminating signal (such as a stimulus artifact).

Result:
Returns compressed wave as value of XFCN.

Example:
```
global wave1,wave2
put 0 into resultType  -- same type as wave1
-- subtract wave2 from wave1, result goes in wave1
put SubtractWaves ("wave1","wave2",resultType) into wave1
```

TabToComma

Type: XFCN

Syntax:
`TabToComma (theData)`

Description:
This XFCN converts a variable (`theData`) from tab-delimited to comma-delimited format. Use this XFCN to convert tab-delimited tables to HyperCard format (comma-delimited). Note that the variable `theData` is passed by *value* (no quotes), and not by name. You will commonly call `GetScrap` before `TabToComma` to get the contents of the clipboard.

Result:
Returns as the value of the function, the data with all tab characters changed to commas. Existing commas are left unchanged.

Example:
```
-- make a tab-delimited table
put "1" & tab & "2" & tab & "3" & return into x
put "4" & tab & "5" & tab & "6" & return after x
put TabToComma(x) into x  -- convert to comma-delimited
format
```

`x` is passed to `TabToComma` by value, therefore it is not enclosed in double quotes. The contents of `x` will be:

```
1,2,3
4,5,6
```

where each item in a row is separated by a comma.

ThresholdWave

Type: XFCN

Syntax:
```
ThresholdWave
("theWave",threshold,hysteresis,resultType)
```

Description:
Threshold detects the wave in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global):

$$y_n = \begin{cases} 1, \text{ if } w_n \geq \text{threshold} \\ 0, \text{ if } w_n < \text{threshold} \end{cases}$$

where:

n ranges from 0 to `npoints`-1

w_n are points in the `theWave`

y_n are points in the result

In practice, noisy waves could cause several apparent threshold crossings where only one is desired. The actual threshold is adjusted by ±`hysteresis`/2 with each crossing. This is a very useful function, especially when combined with other operations. For example, differentiating and thresholding a wave at the zero level will detect the peaks and troughs.

Result:
Returns compressed wave (containing only zeroes and ones) as the value of the XFCN.

Example:
Create a noisy sine wave and threshold detect without hysteresis. The result is scaled up and offset so it can be compared with the original sine wave:

```
global w0,w1

put 2048 into npoints
put 1500 into pkAmpl
put 2 into cycles
put 0 into phase
put 0 into offset
put -12 into resultType

put 0 into hysteresis
put 0 into threshold
-- make a sine wave
put
InitWaveSin(npoints,cycles,pkAmpl,phase,offset,resultTy
pe)¬
into w0
-- add some noise
put InitWaveNoise (npoints,pkAmpl/10,offset,resultType)
into w1
put AddWaves ("w0","w1",0)  into w0
-- threshold detect, result into w1
put ThresholdWave
("w0",threshold,hysteresis,resultType) into w1
-- scale and offset result
put MultWaveK("w1",2000,0) into w1
put AddWaveK("w1",-1000,0) into w1
```

The result from this example is shown in Fig 11-13a. Note how noisy the threshold crossings are due to the added white noise.

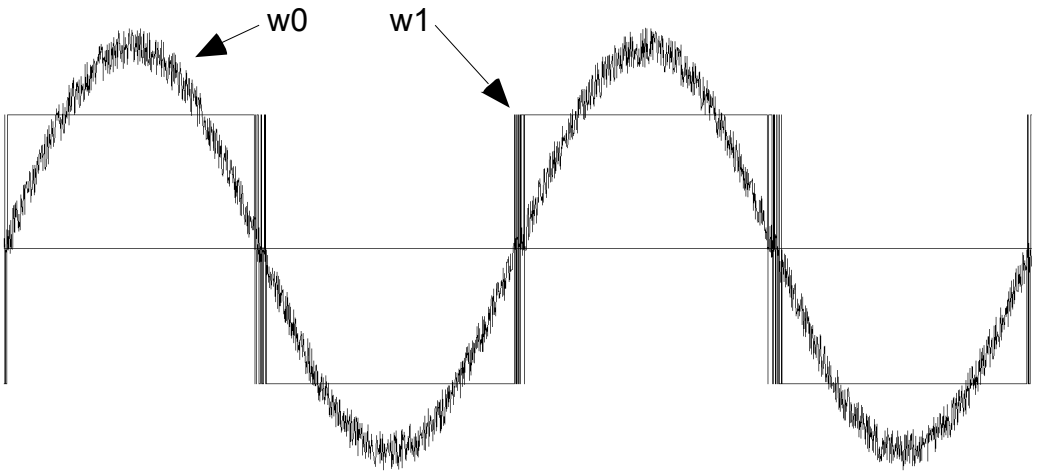


Fig.11-13a: Threshold detection of noisy sine wave with hysteresis = 0. The detection was noisy with several apparent crossings reported with each swing of the sine wave.

The same threshold detection with hysteresis = 300 (300/1500 = 20 % of peak amplitude) is shown below:

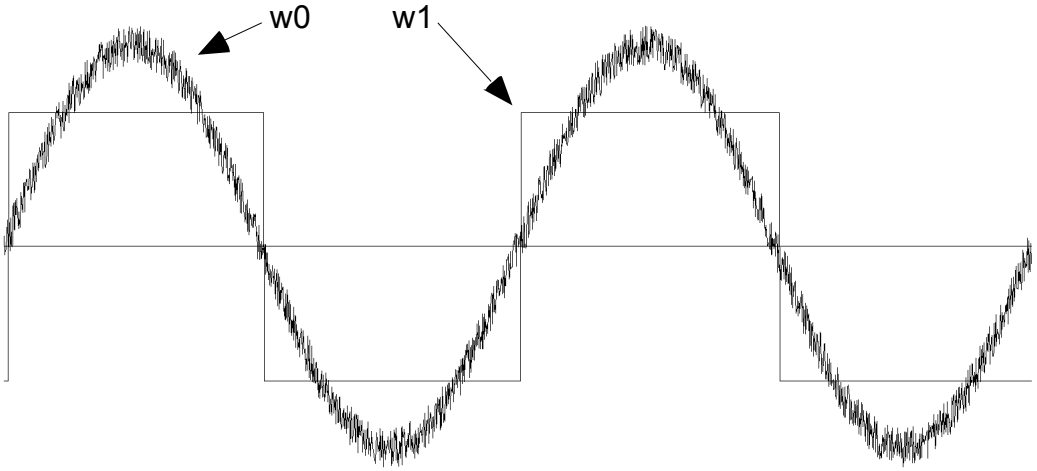


Fig.11-13b: Threshold detection of same noisy wave with hysteresis = 20 % of peak amplitude of sine wave. Here the crossings are clean and the detector is immune to the noise on the sine wave.

The crossings are now clean, but the exact point in time is less certain. You will have to adjust the hysteresis to best suit your signal.

Trim

Type: XFCN

Syntax:
```
Trim ("theWave",sampleInterval,segment)
```

Description:
Deletes a number of points from the wave in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global), corresponding to a segment `segment` μs long. If `segment` is negative, deletes points from beginning of wave, otherwise deletes from end of wave. The result is placed back into `theWave`.

This function is useful if you acquire a wave that is longer than you need. For example, you might need to generate a long conditioning pre-pulse with `AcqWaveDAC`, but are only interested in the signal evoked by the shorter pulse. Use `Trim` to remove the segment of wave corresponding to the pre-pulse time.

Result:
Returns trimmed wave in same global as original. Wave type is same as original. The value of the XFCN equals the number of points remaining in the trimmed wave.

Example:

```
global theWave

put 25 into sampleInterval
put 8192 into npoints

put 0 into DACchannel    -- the D/A channel, 0 or 1
put -5000 into DACpre    -- analog level of pre-pulse
(in mV)
put 50000 into prePulse  -- long pre-pulse (in µs)
put 7000 into DACpulse   -- analog level during pulse
(in mV)
put 1000 into pulseWidth -- short pulseWidth (in µs)
put 0 into DACpost       -- final analog level after
pulse (in mV)

put 0 into startMUX      -- the A/D channel
put startMUX into endMUX

-- adjust pulseWidth and prePulse to multiples of
sampleInterval
put round(pulseWidth/sampleInterval)*sampleInterval
into pulseWidth
put round(prePulse/sampleInterval)*sampleInterval into
prePulse

-- acquire the wave and generate analog pulse
AcqWaveDAC sampleInterval,npoints,startMUX,endMUX,¬
DACchannel,DACpre,DACpulse,DACpost,prePulse,pulseWidth,
"theWave"

-- delete the segment acquired during the pre-pulse
put Trim("theWave",sampleInterval,-prePulse)
```

The message box reads '6192', the number of points in the trimmed wave. This means that 2000 points were deleted from the beginning of the wave corresponding to 2000 * 25 (`sampleInterval`) = 50000 µs, the length of the pre-pulse.

WaveToEventList

Type: XFCN

Syntax:
`WaveToEventList ("theWave",sampleInterval)`

Description:
Returns a list of *events* from the wave in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global). `sampleInterval` is the sampling interval of the original wave in µs per point.

There are two types of events: a positive zero-crossing (when the signal crosses from ≤ 0 to > 0) is a 1 event. A negative zero-crossing (when the signal crosses from > 0 to ≤ 0) is a 0 event. Events are reported as 2 comma-delimited items on each line of a list. The first item is the type of event (0 or 1) and the second is the time the event occurred (= point number * `sampleInterval`). If you want just the point numbers when an event occurred, rather that the time, pass `sampleInterval` = 1.

This function is very useful for extracting the times when certain events occurred. You can use this XFCN with other operations such as filters, differentiators and threshold detectors to build very powerful signal analysis functions. The 'Peak Detector' button is an excellent example of how to build sophisticated commands.

Result:
Returns a table of events (0 or 1) and the time the event occurred, separated by a comma. The number of lines in the list equals the number of events detected. A maximum of 400 events can be reported.

Example:
Generate a cosine wave and pass it to `WaveToEventList`:

```
global theWave

put 25 into sampleInterval -- assume
put 2048 into npoints
put 1500 into pkAmpl
put 1 into cycles
put 90 into phase          -- make a cosine
put 0 into offset
put "F" into resultType    -- float type

put InitWaveSin
(npoints,cycles,pkAmpl,phase,offset,resultType)¬
into theWave
-- get the event list
get WaveToEventList ("theWave",sampleInterval)
```

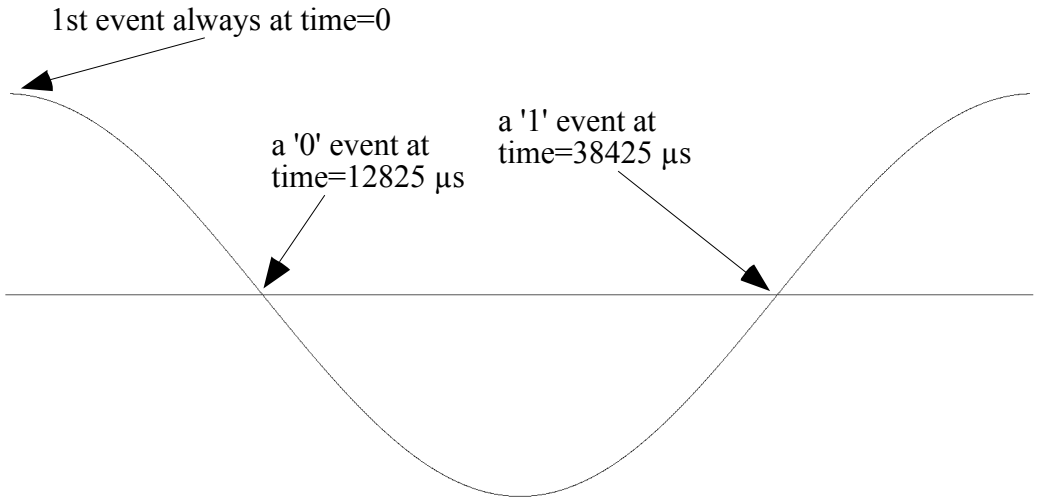


Fig. 11-14: One cycle of a cosine has three events.

Fig.11-14 shows the wave generated by this script. Every wave always has one event at time 0 (1 if first point > 0, or 0 if first point ≤ 0). The cosine example has two additional events at the times shown. The variable `it` will contain the following table:

```
1,0
0,12825
1,38425
```

representing the type of event and the time it occurred.

WaveToXYTable

Type: XFCN

Syntax:
`WaveToXYTable ("theWave",leftX,rightX,topY,bottomY)`

Description:
This XFCN is similar to `CopyXYTable`, except that the result is returned as the value of the function rather than being copied to the clipboard. `WaveToXYTable` converts the wave stored in global `theWave` (double quotes are included to remind you to pass the *name* of the global) into a comma-delimited ASCII table. Both X and Y values are converted and copied. The X values are linearly mapped from their point numbers such that the first X value will be `leftX` and the last will be `rightX`. The **XYCoordXformat** global determines the format of the real X values. If you pass zero for both `leftX` and `rightX`, X values will simply be point numbers (i.e. 0, 1, 2 . . npoints-1). The conversion format will default to "%.0f" (see the chapter on WaveTrak globals for an explanation of conversion formats specified by **XYCoordXformat** and **XYCoordYformat** globals).

Y values of integer waves are linearly mapped such that the maximum value (e.g. 2047 for a signed 12 bit wave) will be `topY` and minimum value (e.g. -2048) will be `bottomY`. If you pass zero for both `topY` and `bottomY`, integer Y values are converted without scaling with a conversion format of "%.0f" (i.e. you will get integer values ranging from -2048 to 2047 for a signed 12 bit wave). Because there are no full scale limits for float waves, Y values are always converted without translation. The **XYCoordYformat** global determines the format of the real Y values.

Use this XFCN to convert waves to numerical tables if you need to accurately determine the values of individual point numbers. Remember that point n in `theWave` will be in line n+1 in the table returned by this function.

Result:
Translates the wave into a comma-delimited table of X-Y data pairs and returns the table as the result of the function.

Example:
This is an example of what you would get if you converted the sample sine wave in the first trace card. The code fragment assumes that all wave descriptors have been initialized when you opened the trace card:

```
global theWave,leftX, rightX, topY, bottomY
global XYCoordXformat,XYCoordYformat
-- save globals
put XYCoordXformat into tempX
put XYCoordYformat into tempY
-- 3 digits after the decimal point
put "%.3f" into XYCoordXformat
put "%.3f" into XYCoordYformat
get WaveToXYtable
("theWave",leftX/1000,rightX/1000,topY/1000,¬
bottomY/1000)
-- restore globals
put tempX into XYCoordXformat
put tempY into XYCoordYformat
```

Note that we elected to convert the X-Y values in ms and volts, rather than µs and mV, by passing `leftX/1000`, `rightX/1000`, `topY/1000`, `bottomY/1000` instead of `leftX`, `rightX`, `topY`, `bottomY`. To make sure that we get enough precision, the conversion formats were reset to 3 digits after the decimal point (`"%.3f"`). Note that we saved, then *restored* the values of **XYCoordXformat** and **XYCoordYformat** globals to avoid interfering with other WaveTrak functions. The variable `it` will contain the following table consisting of 2048 lines (= `npoints`):

```
  0.000,-0.364
  0.025,-0.291
  0.050,-0.208
      .
      .
      .
  51.150,-0.696
  51.175,-0.603
```

WaveToYTable

Type: XFCN

Syntax:
```
WaveToYTable ("theWave",topY,bottomY)
```

Description:
This XFCN is similar to `CopyYTable`, except that the result is returned as the value of the function rather than being copied to the clipboard. `WaveToYTable` converts the wave stored in global `theWave` (double quotes are included to remind you to pass the *name* of the global) into an ASCII table. Only Y values are converted. Y values of integer waves are linearly mapped such that the maximum value (e.g. 2047 for a signed 12 bit wave) will be `topY` and minimum value (e.g. -2048) will be `bottomY`. If you pass zero for both `topY` and `bottomY`, integer Y values are converted without scaling with a conversion format of "%.0f" (i.e. you will get integer values ranging from -2048 to 2047 for a signed 12 bit wave). Because there are no full scale limits for float waves, Y values are always converted without translation. The **XYCoordYformat** global determines the format of the real Y values.

Use this XFCN to convert waves to numerical Y values if you need to accurately determine the values of individual point numbers. Remember that point n in `theWave` will be in line n+1 in the table returned by this function.

Result:
Translates the wave into a column of Y data and returns the table as the result of the function.

Example:
This is an example of what you would get if you translated the sample sine wave in the first trace card. The code fragment assumes that all wave descriptors have been initialized when you opened the trace card:

```
global theWave
get WaveToYtable ("theWave",0,0)
```

Here we elected to convert the Y values as raw integers without scaling, by passing zero for both `topY` and `bottomY`. Any conversion format in the **XYCoordYformat** global was ignored and defaulted to "%.0f". The variable `it` will contain the following column of data, consisting of 2048 lines (= `npoints`):

```
-75
-60
-43
 .
 .
 .
-143
-124
```

Window

Type: XFCN

Syntax:
```
Window
("theWave",windowType,offset,direction,resultType)
```

Description:
Multiplies (`direction` = 1) or divides (`direction` = -1) the wave in global variable `theWave` (double quotes are included to remind you to pass the *name* of the global) by a window function. The window function is selected by `windowType`:

1: Hanning
2: Parzen (triangular)
3: Welch

`offset` is first subtracted from the wave before windowing, then added back. Fig.11-15 illustrates how the offset parameter is useful. If a sine wave is floating on a DC level and a window is applied, the result shown in a) is obtained, which is not what we want. Instead, if the DC level is temporarily removed while the sine wave is windowed, we get the desired result. `resultType` selects the data type of the result (e.g. -12 for a signed 12-bit integer wave, "F" for a single-precision floating point wave). Passing `resultType` = 0 will return same type as `theWave`. If an integer wave is requested by `resultType` and some elements are out of range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value.

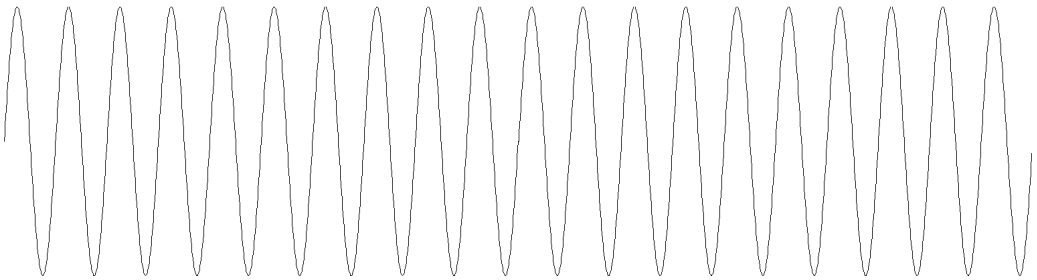
Window functions are useful for tapering a wave at either end to avoid 'spectral leakage' during FFT computations. The `direction` parameter allows you to 'de-window' a wave; this is useful if you want to filter a wave with the FFT while applying a window. Remove the window by calling `Window` a second time with `direction` = -1. Floating point round-off errors usually produce spurious transients at the extremes of de-windowed signals. Experiment with various parameters to get the best results (see the 'Lo/Hi-pass + window' buttons for scripting examples of how windows can be used in digital filtering).

Result:
Returns compressed wave as value of XFCN.

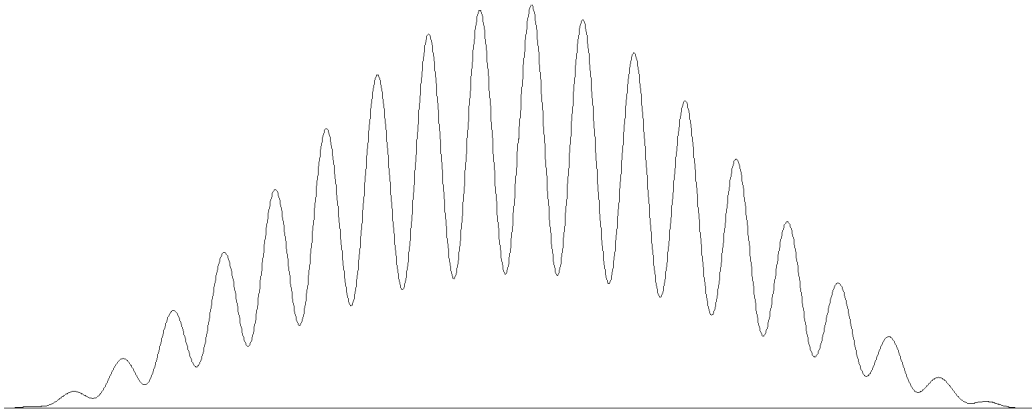
Example:

```
global theWave

put 2048 into npoints
put 500 into pkAmpl
put 20 into cycles
put 0 into phase
put 1000 into offset    -- sine wave floats on a DC
level
put -12 into resultType

put InitWaveSin
(npoints,cycles,pkAmpl,phase,offset,resultType)¬
into theWave

put 1 into windowType  -- Hanning window
put 0 into offset      -- window without offset
compensation
-- (or put 1000 into offset for result in (c), below)
put 1 into direction
put Window
("theWave",windowType,offset,direction,resultType)¬
into theWave
```

a)



b)



c)

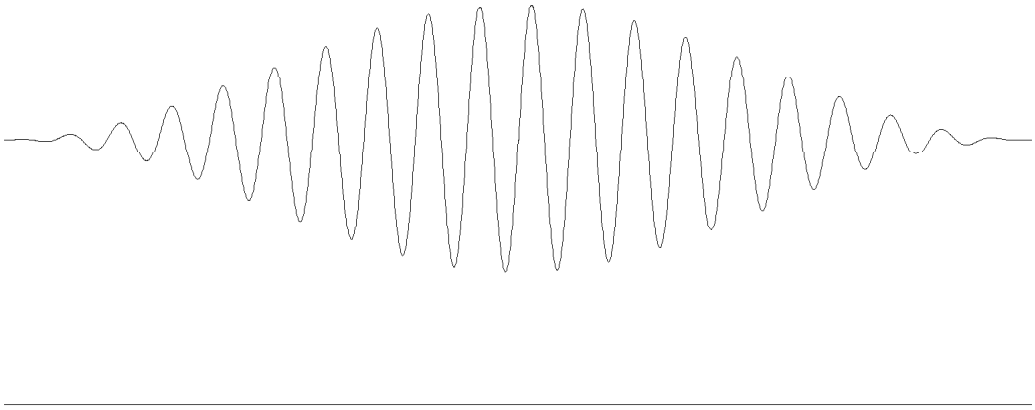


Fig.11-15: If a signal floats on a DC level (a) and is multiplied by a window function without offset compensation, the ends will be tapered to zero, rather than to the DC level (b). By passing the DC level in the offset parameter, the window function is correctly applied (c).

WriteDAC

Type: XCMD

Syntax:
`WriteDAC DACchannel,DAClevel`

Description:
This XCMD outputs an analog level (`DAClevel`, in mV, must be an integer) at the output of one D/A converter (`DACchannel`, 0 or 1). This XCMD uses the globals **DACmin**, **DACmax** and **DACbits** to determine the correct binary word to write to the D/A converter in order to output the requested voltage. Any gain external to the D/A converter is not taken into account, and you must scale `DAClevel` accordingly before calling `WriteDAC` (see example).

Result:
None.

Example:

```
global DACGainTable     -- contains external gain
information

put 0 into DACchannel   -- the D/A channel (0 or 1)
put 1000 into DAClevel  -- the analog output voltage
(mV)

-- adjust DAC level w.r.t external DAC gain
get line (DACchannel+1) in DACGainTable
put round(DAClevel/it) into adjDAClevel

-- output the analog level
WriteDAC DACchannel,adjDAClevel
```

WriteModeByte

Type: XCMD

Syntax:
```
WriteModeByte theByte
```

Description:
This XCMD writes `theByte` (0 to 255) to the mode register on the MacADIOS II card. A value of 0 corresponds to a binary bit pattern of 00000000, and 255 to 11111111. Consult the MacADIOS II hardware manual for details on what each bit does in the mode register. You can use this XCMD to reset any bits that were cleared by `SetADGain`.

Result:
None.

Example:

```
put 2 into theByte
WriteModeByte theByte
```

Sets the programmable gain amplifier to 100.

WriteTTLbit

Type: XCMD

Syntax:
`WriteTTLbit bitNumber,theLevel`

Description:
This XCMD sets (`theLevel` = 1) or clears (`theLevel` = 0) bit `bitNumber` (0 to 7) of TTL output port. By presetting the level of a bit, you can determine the polarity of pulses for functions that toggle TTL bits (e.g. `AcqWaveTTL`).

Result:
None.

Example:

```
put 2 into bitNumber
put 1 into theLevel
WriteTTLbit bitNumber,theLevel  -- set bit 2 (third
bit)
.
.
AcqWaveTTL sampleInterval,npoints,startMUX,endMUX,
bitNumber,preTrig,pulseWidth,gList
```

By setting bit 2, `AcqWaveTTL` will generate an active low TTL pulse. It's always a good idea to preset the level of toggled bits to ensure the correct polarity. WaveTrak *clears* all output bits at startup.

WriteTTLbyte

Type: XCMD

Syntax:
```
WriteTTLbyte theByte
```

Description:
This XCMD writes `theByte` (0 to 255) to the TTL output port. A value of 0 corresponds to a binary bit pattern of 00000000, and 255 to 11111111. All 8 bits of the port are affected.

Result:
None.

Example:

```
put 255 into theByte
WriteTTLbyte theByte
```

All 8 output bits will be TTL high. WaveTrak uses `WriteTTLbyte` to clear all output bits at startup.

XYTableToWave

Type: XFCN

Syntax:
`XYTableToWave (XYTable,"theWave",resultType)`

Description:
This XFCN is the key function for importing data into WaveTrak. The input in `XYTable` is an ASCII table of numerical values, assumed to be equally spaced in time (or any other dimension). This function will convert the table and return a compressed WaveTrak wave in the global variable `theWave` (double quotes are included to remind you to pass the *name* of the global).

Although the name of the function suggests that tables of XY values are used for input, tables of only Y values are recognized as well. If the first line contains a delimiter, i.e. a comma, tab or any number of spaces, an XY table is assumed. If no delimiters are found, a Y table is assumed. Every line must be terminated with a carriage return. If an XY table is detected, the function returns as the sample interval the difference between the first two X values (see Result, below). If a Y table is detected, the sample interval is undefined and the function returns 0; you will have to determine what the original sampling rate was yourself.

`resultType` tells the function what type of wave you would like returned (e.g. -12, signed 12-bit binary; F, float; etc...). If an integer wave is requested, values will be rounded. If one or more values is beyond the requested integer range, a silent overflow error (**XCMDErr** = 42) is returned and the out-of-range elements are clipped to the maximum or minimum allowable value; the remainder of the conversion is completed.

Result:
Two-item, comma-delimited list:
1st item: sample interval, either the difference between the first two X values for an XY table, or 0 for a Y table.
2nd item: number of points in converted waves (which equals the number of lines in ASCII table).

Example 1, convert clipboard contents into a wave:

Assume that you pasted the following table of values from another application onto the clipboard (each line ends with a carriage return):

```
-363.9
-290.6
-207.6
-134.3
   .
   .
   .
5277.2
5296.7
5306.5
5326.0
5335.8
```

The table consists of a total of 2048 entries (or lines). Executing the following script would yield the following results:

```
  global XCMDErr, theWave
  put "F" into resultType  -- convert floating point
values

  -- copy the clipboard into a variable
  put GetScrap() into XYTable

  -- translate Y values into a WaveTrak wave
  get XYTableToWave (XYTable,"theWave",resultType)
  put it into XYresult
```

No delimiters are present so the table is assumed to contain Y-values only. The global variable `theWave` will contain the compressed floating point values converted from the table; this wave will have 2048 points. `XYresult` will contain "0,2048", signifying that the sampling rate could not be determined and that there were 2048 points converted.

Example 2, convert a disk file into a wave:

You have a text file containing the following ASCII data (each line ends with a carriage return):

```
0,-363.9
10,-290.6
20,-207.6
30,-134.3
    .
    .
    .
50920,5277.2
50930,5296.7
50940,5306.5
50950,5326.0
50960,5335.8
```

The table consists of a total of 2048 entries (or lines). Executing the following script would yield the following results:

```
global XCMDErr, theWave
-- convert signed 12-bit integer values
put -12 into resultType

-- select the file
answer file "Select the input file:" of type TEXT
put it into fName
-- read entire file into fBuffer
put empty into fBuffer
open file fName
repeat
  read from file fName for 16834
  if it is empty then exit repeat
  put it after fBuffer
end repeat
close file fName

-- translate X,Y values into a WaveTrak wave
get XYTableToWave (fBuffer,"theWave",resultType)
put it into XYresult
```

Delimiters are present (commas) so the table is assumed to contain X,Y pairs. The global variable `theWave` will contain the compressed 12-bit signed integer values converted from the table; this wave will have 2048 points. Note that all values will be rounded to integers, and that any values less than -2048 or greater than 2047 will be clipped to these limits; that is the last several lines shown in the sample text will all be limited to 2047, as this is the maximum range that can be represented by a signed 12 bit integer. The global `XCMDErr` is set to 42 indicating that an overflow occurred. `XYresult` will contain "10,2048", signifying that the sampling interval was calculated as 10, and that there were 2048 points converted.

In Summary

- The WaveTrak data acquisition and digital signal processing toolbox consists of over 70 highly optimized XCMDs and XFCNs.

- You invoke XCMDs and XFCNs much as you would normal HyperTalk commands and functions, respectively. The parameter list must be enclosed in parentheses when calling XFCNs, but not when calling XCMDs.

- XFCNs return a single result as the value of the function (although they may additionally return multiple results in global variables).

- When passing global variables to receive data from an XCMD/XFCN, always pass the *name(s)* of the globals, either as quoted literals, or as variables containing the name(s) of the global variables.

- All XCMDs/XFCNs report errors in the global **XCMDErr**. A value of zero means no error occurred.