Scripting with the WaveTrak Toolbox

Earlier chapters introduced some of the fundamentals of program flow and data storage techniques used by WaveTrak. This chapter will give you more detailed information about how WaveTrak goes about converting and transferring data from an external signal source to a trace card in your stack. Although most of the examples concern A/D conversion with the A/D version of WaveTrak, the concepts are identical for waves imported from disk or the clipboard. Once you understand the basics, you will be able to create your own scripts easily, since all of the difficult aspects of data acquisition and organization are taken care of automatically. This chapter assumes that you have a working knowledge of HyperTalk. You should be familiar with concepts such as handlers, messages and global variables. You should also be aware how messages are passed up the HyperCard hierarchy and how to edit scripts. If any of the standard HyperCard commands mentioned in this chapter are unfamiliar to you, please consult your HyperCard reference manuals.

Waves and Data Types

In WaveTrak, a *wave* is defined as a series of points regularly spaced in time (or any other dimension you define). The points (or elements) of a wave are held in an *array*, which is a block of contiguous memory locations in RAM containing the values of the points (the y-values). Since the points are always regularly spaced in time, the x-values (time) are not stored but derived from the point number (the *index*) and the original sampling interval. Unlike the numbering convention in HyperCard, points in WaveTrak waves are numbered beginning with zero up to one less than the number of points in the wave. For example, if you have a wave with 1024 points, the point number of the first sample is always zero, and the last point would be numbered 1023. The first point of any wave always corresponds to

0 μs; at a sampling interval of 10 μs/sample (100 kHz), the last point in the example would correspond to time = 10230 μs = 10.23 ms. Since each point actually represents a signal over a short stretch of time equal in duration to the sample interval, the total sample window will be 10.24 ms in duration. Fig. 9-1 illustrates this important point:

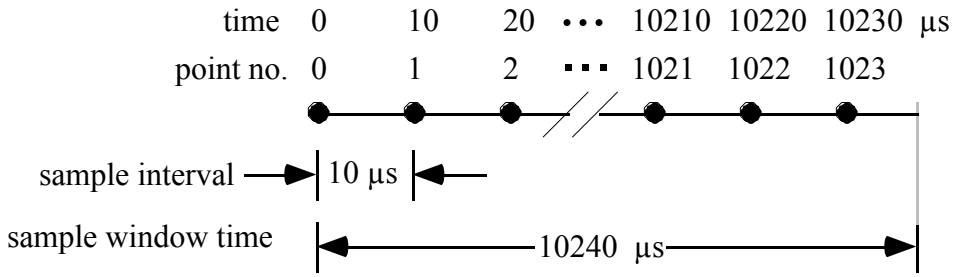

Fig.9-1: relationship between the sample interval, point number (or index) and the total sample window time.

Note that in HyperCard, items and lines in a list are numbered beginning with 1.

The elements in a wave can be represented either as integers (up to 16 bits) or single precision (32-bit) floating point numbers; the *type* of a wave is determined by which representation is used for its elements. Values read from the A/D converter are always integers. The on-board converter on the MacADIOS II board has 12 bits of resolution and can be configured as straight binary (positive values ranging from 0 to 4095) or two's complement (positive and negative values ranging from -2048 to 2047). A wave consists of more than just a series of elements, but also contains a header identifying the data type (floating point or integer, and if integer, how many bits of resolution), how many points it has as well as other information used internally. You can find out the type of a wave by calling the function `getWaveType` in the stack script. Table 1 lists some of the more common WaveTrak data types.

Table 1: list of codes used to identify common WaveTrak data types. Integer types are identified by the number of bits of resolution; this number is negated if the type is a signed integer.

Code	**Data type**
F	32-bit single precision floating point.
-12	12-bit signed (2's complement) binary integer (-2048 to 2047).
12	12-bit unsigned binary integer (0 to 4095).
-16	16-bit signed (2's complement) binary integer (-32768 to 32767).
16	12-bit unsigned binary integer (0 to 65535).

Many XCMDs that generate waves mathematically or operate on existing waves give you the option of setting the result type by passing a code; if you pass zero, the result will be of the same type as that of the wave you passed to the XCMD.

Example 1:

```
global theWave
AcqWave sampleInterval, npoints, startMUX, endMUX,"theWave"
put getWaveType (theWave) -- no quotes, wave is passed by value
```

The `AcqWave` XCMD acquires a wave from the 12-bit A/D converter. Assuming that it is jumpered for 12-bit two's complement coding, line 3 will write '-12' in the message box. Each element in `theWave` will be a signed 12-bit integer between -2048 and 2047. `getWaveType` requires that you pass waves by value, that is, without enclosing the variable name containing the wave in quotes.

Example 2:

```
global theWave
AcqWave sampleInterval, npoints, startMUX, endMUX,"theWave"
put "F" into resultType
put 0 into K
put AddWaveK ("theWave",K, resultType) into theWave
```

Here, a 12-bit wave is again acquired, and a constant is added to each element. The result still ranges from -2048 to 2047, but will be a floating point type, because the `resultType` parameter was set to "F" and forced `AddWaveK` to return a floating point wave. You can use this technique (adding zero) to convert from one type to another.

Example 3:

```
global w0,theWave
AcqWave sampleInterval, npoints, startMUX, endMUX,"theWave"
put 0 into resultType
put AddWaves ("w0","theWave",resultType) into w0
```

`w0` was created elsewhere and was sent to you as a global. All you want to do is to add a newly acquired wave to `w0`. You don't know the data type of `w0`, and you don't want to change it. Passing zero in `resultType` will return the same type as the first wave (`w0`) in the `AddWaves` parameter list. *Use a code of zero to preserve the original type*. This avoids the extra step of having to find out the wave type first. In fact, you will use a code of zero most of the time unless you have a specific reason for changing the data type of a wave.

Technical note:

Users familiar with HyperTalk and more traditional programming languages are probably wondering how raw integer or floating point arrays can be stored in HyperCard variables and fields. The problem is that HyperCard data structures are always null-terminated, and assume that only ASCII values will be stored in them. Therefore, anytime your raw data would have a zero byte, and you simply copied it to a HyperCard variable, this would signal the end of your data and HyperCard would ignore the rest. WaveTrak therefore performs an additional step where it encodes and compresses your raw data into a HyperCard-compatible format. Typically a 1000-point, 12 or 16 bit integer wave can be stored in a little more than 1000 bytes, a 2:1 compression. Floating point waves are not compressed, but must be encoded, resulting in a size slightly more than 4 bytes per element. Line 12 of the `GetWaveStats` XFCN contains the size, in bytes, of the encoded waveform.

Scripts

The overall structure of the WaveTrak environment was carefully designed so that power and flexibility is available, but not at the expense of ease of use. By extending the standard HyperTalk language with the WaveTrak data acquisition toolbox, sophisticated operations, such as those described in earlier chapters, can be easily assembled using only HyperTalk scripts. This section describes two basic acquisition buttons, step by step. Once you become comfortable with these two scripts, the best way to learn more is to examine other scripts; comments have been included throughout to explain important points.

Let's go over the script of the 'Single' button, line by line, to see how simple acquisition is done. More complicated paradigms can be easily programmed by starting with this script and adding functions as needed. Below is a listing of the script, with line numbers added so the following discussion can clearly refer to specific parts of the code (note that some of the longer lines [22 for example] will wrap in the listing below, but in fact belong in a single line in the script):

```
1 on mouseUp
2   -- Acquires a single wave from A/D channel selected
3   -- in pop-up menu in trace card.

4   -- Copyright © 1991 Ortex Systems Inc. All rights
reserved.
```

```
5
-------------------------------------------------------------
-
6   -----------------  PARAMETERS
----------------------------
7
-------------------------------------------------------------
-

8   -- none

9
-------------------------------------------------------------
-

10  global HardwareOK,XCMDErr,timeStamp
11  global sampleInterval,npoints,FSTable
12  global theWave

13  put the seconds into timeStamp

14  put bg fld "ADChannelFld" in cd "StdTraceCard" into
startMUX   15  put startMUX into endMUX

16  -- acquire the wave
17  AcqWave sampleInterval, npoints, startMUX,
endMUX,"theWave"

18  -- did an error occur?
19  if XCMDErr=0 then

20    newTrace --  create a new trace card

21    put theWave into bg field "data"  -- store the wave

22    put cd fld "HParamLegends" in cd "SysParams" into bg
fld
     "HParamLegends"
```

```
23    put cd fld "ReadingLegends" in cd "SysParams" into
bg fld
     "ReadingLegends"
24    put cd fld "Readings" in cd "SysParams" into bg fld
     "Readings"

25    -- copy default params
26    get cd fld "HParams" in cd "SysParams"
27    repeat with j=1 to 3
28      put line j in it into line j in bg fld "HParams"
29    end repeat
```

```
30    get line (startMUX+1) in FSTable  -- A/D full scale,
units
31    put it into line 4 in bg fld "HParams"

32  else
33    ErrNum XCMDErr
34  end if

35  send openCard -- plot the wave
36  select after last char of field "commentField"
37end mouseUp
```

Most buttons respond to 'mouseUp' messages, and lines 1 and 37 enclose the handler which intercepts this message when the button is pressed, executing all instructions between `on mouseUp` and `end mouseUp`. Lines 2 to 9 are comments describing what the button does, and what kinds of parameters it expects from you. The Single button has no parameters. Lines 10 to 12 declare all the global variables used in the handler. Globals are necessary for two reasons. First, they contain needed data that was defined elsewhere. For example, **sampleInterval** is updated when you enter a new sampling rate in the Scope card. By placing the value into a global, all scripts in the stack have access to this value. WaveTrak places important values of widespread interest to many scripts into globals. The chapter on WaveTrak Globals describes the standard global variables and what they are used for.

The second reason for using globals is that XCMDs can only return data in a global variable. For example, `theWave` must be declared a global so that the `AcqWave` XCMD (line 17) can find this variable in which to return the digitized result. Line 13 saves the current date and time (`the seconds` is a standard HyperCard command) in `timeStamp` so that the time of acquisition can be later written to the appropriate field in the trace card. Line 14 reads which A/D channel is currently selected in the pop-up menu under the Single button; since we're acquiring only one channel, the starting and ending A/D multiplexer (MUX for short) channels are the same (line 15).

Line 17 calls the XCMD that digitizes the signal; **sampleInterval** (in µs) and **npoints** (the number of points/wave) are globals updated in the Scope card. *One very important point here is that the* name *of the global receiving the data is*

passed (i.e. "theWave" in double quotes), and not the value of the global without the quotes. This holds true for all XCMDs/XFCNs *returning* data in global variables. Here are some examples illustrating this very important point:

Correct

```
  on mouseUp
   global theWave
   ...
   --  pass the name as a quoted string
   AcqWave sampleInterval, npoints, startMUX,
endMUX,"theWave"
   --  or put the name in a variable first, then pass the
variable
   put "theWave" into gName
   AcqWave sampleInterval, npoints, startMUX, endMUX, gName
  end mouseUp
```

In the above segment, the name of the global `theWave` is first passed explicitly as a quoted string. Its name is then copied to a variable (`gName`, which need not be global itself), and the *contents* of `gName` is passed without quotes. Both forms are acceptable.

Incorrect

```
  on mouseUp
   ...
   AcqWave
sampleInterval,npoints,startMUX,endMUX,"theWave"
  end mouseUp
```

In this example, `theWave` was not declared as a global. XCMDs can *only* return values in globals.

Scripting

```
on MouseUp
 global theWave
 ...
 -- theWave passed by value, without quotes: incorrect
 AcqWave sampleInterval,npoints,startMUX,endMUX,theWave
end mouseUp
```

This is also incorrect, because, although `theWave` was declared as a global, it was then passed by value without quotes. XCMDs that return data in globals must receive the *name(s)* of the globals.

> Technical note:
>
> Programmers will recognize this as a way of passing a variable by *reference* rather than by *value*. XCMDs can only return data in global variables and require the name of the global, so they can in turn pass it in a 'SetGlobal' callback. The second example above is incorrect but will not generate an error. Instead, a global variable 'theWave' will be automatically created by the SetGlobal callback. Nevertheless, it is good programming practice to explicitly declare `theWave` as a global in your handler to avoid confusion.

This technique allows you to simulate an array of globals in HyperCard by passing a variable containing a comma-delimited list of global names. See the `DrawWaveCoords` XCMD and the `TrOverlay` handler in the trace card background script for examples of how arrays of waves are implemented in WaveTrak.

All XCMDs and XFCNs report errors in the global **XCMDErr**. Line 19 checks to see if `AcqWave` completed successfully. If no error occurred, all XCMDs return zero in this global. A list of all WaveTrak errors and their meaning appears in a later chapter.

If all went well, line 19 directs the flow of execution towards creating a new trace card by calling `newTrace` in the stack script. The mark of the new card will be the same as that of the previous trace; the first trace under a root defaults to 'A'. Line 19 saves the digitized signal (placed in `theWave` by `AcqWave`) in the background field 'data'. This hidden field stores a single wave in each trace. Because the current version of HyperCard limits the size

of fields to 30000 characters (bytes), this limits the *encoded* size of waves. This translates into about 29500 points for integer waves and about 7400 points for floating point.

> Technical note:
>
> The size of compressed waves will depend on the data. WaveTrak may be unable to compress some very noisy integer waves, and will simply encode them into a HyperCard-compatible format. These waves can require more than two bytes per point and will further limit the number of points that can be saved in the data field. There is no way to predict how large an encoded wave will be. Although fields can only store 30000 bytes, variables are limited only by available RAM. Therefore you can acquire and manipulate very large arrays, but you can only *save* 30000 bytes in a field. In fact, using 32-bit addressing and virtual memory, you can perform calculations on huge arrays containing millions of points, although swapping these large arrays to and from disk will slow processing considerably.

This script does not check the size of `theWave`; if you attempt to save more than 30000 characters in the data field, HyperCard will generate an error. Checking the size of `theWave` is left as an exercise. Fields are automatically written to disk and this is how waves are permanently stored (no need to issue Save commands in HyperCard). This time, you are putting the *contents* of `theWave` (and not its name) into the field 'data', so no quotes here. Lines 22 and 23 copy the default legends from the System Parameters card for the Hardware Parameters and Readings fields, and default Readings are copied by line 24 into the Readings field. You can copy other readings into this field if you wish. For instance, say you take a temperature measurement with each acquisition; you can write over the default value of 37° in line 1 of the Readings field.

Because no digital or analog pulses were delivered with this simple button, only the first three lines of the Hardware Parameters field are relevant, and only the first three default values are copied from the System Parameters card by lines 26 to 29. Note that WaveTrak ensures that the values in the System Parameters card always match the globals (such as **sampleInterval** and **npoints**) so it's safe to take the values either from the Hardware Parameters field or from the globals. The fourth line of the Hardware Parameters field however, will depend on the A/D channel selected; the **FSTable** global holds a copy of the 'Full-scale' column in the 'External A/D Gain' table (see Fig. 6-8). The line corresponding to the selected A/D channel is taken from this global and copied to line 4 of the Hardware Parameters field (1 is added to `startMUX` because A/D channels are numbered 0 to 7, while HyperCard line numbers range from 1 to 8). This completes the acquisition, trace creation and parameter update.

If you acquire other readings from other channels, simply write the appropriate legends and results in whichever line of the Hardware Parameters and/or Readings fields you wish. There is only one restriction: WaveTrak depends on the values in the first four lines of the Hardware Parameters field to correctly draw and scale your wave when you open a new card trace. You must place the correct data into these lines, as illustrated in the Single button.

If an error occurred during the acquisition, line 33 calls the `ErrNum` handler in the stack script which puts up a dialog box with an error message; the message is taken from the field in the ErrorList card.
Line 35 sends an openCard message to display your new wave (as if you just jumped to this card from elsewhere), and line 36 places the text insertion point in the trace comment field so the trace is ready to accept typed comments after the acquisition.

Multiple button:

This script illustrates how multiple channels are acquired and stored, and how arrays of waves are manipulated in WaveTrak. The version shown below assumes that the converter is jumpered for differential input and therefore acquires a maximum of 8 A/D channels. Only those lines that are different from the 'Single' button will be explained in detail, so please refer also to the discussion above:

```
1 on mouseUp
 ------------------------------------------------------
 -----------------  PARAMETERS ---------------------------
 ------------------------------------------------------

 -- channels startMUX to endMUX inclusive will be sampled
2   put 0 into startMUX
3   put 7 into endMUX

 ------------------------------------------------------
```

```
4   global XCMDErr,timeStamp
5   global sampleInterval,npoints,FSTable
6   global w0,w1,w2,w3,w4,w5,w6,w7

7   put the seconds into timeStamp

 -- put global NAMES into a list
8   put "w0,w1,w2,w3,w4,w5,w6,w7" into gList
 -- create a corresponding list of marks
9   put "0,1,2,3,4,5,6,7" into markList
10  put endMux-startMUX+1 into nChannels

11  put "Acquiring..."
12  AcqWave sampleInterval*nChannels, npoints, startMUX,
    endMUX,gList

 -- did an error occur?
13  if XCMDErr=0 then

14    put cd fld "HParamLegends" in cd "SysParams" into
       HParamLegends
15    put cd fld "HParams" in cd "SysParams" into HParams
16    put cd fld "ReadingLegends" in cd "SysParams" into
       ReadingLegends
17    put cd fld "Readings" in cd "SysParams" into Readings

18    put "Writing " & nChannels & " traces to disk..."
19    repeat with traceCtr=1 to nChannels
20      newTrace
21      get item (traceCtr+startMUX) in markList
22      put it into bg fld "Mark"

23      put the value of (item traceCtr in gList) into bg field
        "data"

24      put HParamLegends into bg fld "HParamLegends"
25      put ReadingLegends into bg fld "ReadingLegends"
26      put Readings into bg fld "Readings"
```

```
27      repeat with j=1 to 2
```

```
28        put line j in HParams into line j in bg fld "HParams"
29       end repeat

30       put sampleInterval*nChannels into line 3 in bg fld
        "HParams"
31       get line (startMUX+traceCtr) in FSTable
32      put it into line 4 in bg fld "HParams"

33      send openCard
34     end repeat

35   else
36     ErrNum XCMDErr
37  end if

38  send openCard -- plot the wave
39   put empty
40   select after last char of field "commentField"
41 end mouseUp
```

In all Multiple-type buttons, the A/D channels to be sampled are entered as starting and ending channels (inclusive) in the parameter block (lines 2 and 3). Note that channels must be contiguous, so that you cannot request that only channels 1, 3 and 5 be sampled. The results from each channel must be returned in global variables and these are declared in line 6.

Tip:

For globals that will receive waves, it's a good idea to use the same names in all your scripts (`w0`, `w1`, `w2`, ... in this example). If you make up 8 different names each time you create a Multiple-type button, HyperCard will need to set aside additional memory for each unique name, which could slow down performance or cause you to eventually run out of memory. WaveTrak declares 16 standard globals named `w0` to `w15` which you should use to store waves.

Line 8 illustrates the important concept of setting up wave arrays. We saw that another way of passing a global name to an XCMD is to first put the name into a variable, then pass that variable by value. This method makes it even easier to pass several globals by putting their names into a comma-delimited list. You will notice that line 8 duplicates the names of the 8 globals declared previously (no spaces anywhere) and puts this list into a variable, `gList`. WaveTrak XCMDs have been designed to expect multiple globals as comma-delimited items, consistent with HyperCard's convention of separating items in a list. Line 9 assigns a mark to correspond to each global in the list. Each mark will be stored in the corresponding 'Mark' field when the trace card is created.

The number of A/D channels is computed in line 10 (+1 because `startMUX` to `endMUX` is inclusive). Line 11 alerts the user in the message window that the acquisition is about to begin, and line 12 calls `AcqWave` to do the work. There are two differences in this call compared to the one in the Single button. First, WaveTrak acquisition XCMDs have been designed to maintain the requested sampling rate for each channel when multiple channels are acquired. For instance, if you call `AcqWave` with only one channel to be sampled at a rate of 100 µs/sample, then call it again with two channels, again requesting a sample rate of 100 µs/sample, both channels will be sampled at 100 µs/sample. That is, the time between two samples from the first channel will be 100 µs, as will be the time between two samples from the second channel. The converter therefore needed to sample data at an effective average rate of 50 µs/sample to conserve the requested rate for each of the two channels.

> Technical note:
>
> WaveTrak performs synchronous acquisition (or as close to synchronous as the MacADIOS II hardware will allow). Therefore, with two channels in the example above, acquisitions from the first and second channels would actually be 7 µs apart, with a period of 100 µs between pairs of samples. See the chapter on WaveTrak XCMDs for details about acquisition timing (Fig. 11-1).

Therefore the effective sampling rate demanded of the converter will increase with the number of channels requested. If you select a fast rate for a single channel, you could exceed the converter specification if you then request the same rate for many channels. The Multiple buttons avoid this by scaling down the per channel sampling rate by the number of channels requested (`sampleInterval*nChannels`).

The second important difference is that the global names are passed in the variable `gList`, rather than as a quoted string. Passing the names in a variable may not be as intuitive, but is much more flexible because the script itself can modify which globals are passed simply by changing the contents of `gList` (see the `TrOverlay` handler in the trace background script for a good example).

Tip:

Don't worry if you pass 8 global names in `gList` but sample less than 8 channels; `AcqWave` and related XCMDs will simply ignore the other global names. Also don't worry if you start sampling with channel 4, 5 ... for example. Although results will always be returned starting in `w0`, `w1`... , the data will be correctly extracted in the loop beginning at line 19 and the proper marks assigned.

Line 13 checks if `AcqWave` was successful, and if so, proceeds to create the traces. Lines 14-17 simply copy the default fields into variables, since accessing a local variable is faster than reading from a field in another card. Line 18 informs the user that the data are being written to disk; a repeat loop (line 19) is the most efficient way to cycle through the results. After each new trace is created (line 20), the appropriate mark is extracted from `markList` by offsetting the item number by `startMUX` (line 21). This is needed so that channel ranges that don't begin at zero will retrieve the correct marks. Line 22 overwrites the default mark entered by `newTrace`.

Line 23 illustrates another important aspect of wave arrays. We know that we have our data in globals `w0`, `w1`, `w2` ..., but how do we get these values back in a repeat loop? `(item traceCtr in gList)` will take on the values "w0", "w1", "w2" ... as the loop iterates, but we don't want the *names* of the globals, but rather their *contents* (i.e. the waves they contain). The standard HyperCard function `the value of` does what we need by returning the *contents* of a variable when you pass it the variable *name*. The contents are then copied to the 'data' field, which will be a fresh background field for each new trace card.

Lines 24-29 copy default parameters and readings into each new trace; the effective sampling interval was scaled down by the number of channels (line 30). The correct full scale information is retrieved from **FSTable** by offsetting the line number with `startMUX`. Line 33 will force each new wave to be drawn as it's stored to give the user

visual feedback; you can remove this line to increase execution speed. The remaining instructions perform the same function as in the Single button script.

These two buttons serve as a foundation upon which all other acquisition buttons are built in WaveTrak. See the later chapters for a detailed description of `AcqWave` and related XCMDs.

You may have noticed that scripts can be placed in a number of locations within a stack, including buttons, cards, backgrounds and the stack script itself. How far up the hierarchy you place a handler will depend on how popular that function is within your stack. We have found that it is most intuitive to put the instructions that a button executes right in that button's script. Eventually, you will no doubt write more complex scripts that can be called from many places in your stack. We recommend that you create a blank stack with a single card (using 'New Stack...' *not* 'New WaveTrak Stack' in 'File' menu) and place your custom code in the script of the new stack. Add a `start using myCustomStackName` instruction in WaveTrak's `openStack` handler to add your new custom stack to the message hierarchy. This is how WaveTrak accesses its A/D and DSP libraries. There are two advantages to using your own stack to hold your scripts. First, the existing WaveTrak stack script is already quite large, and there is a 30000 character limit on the size of any single script. By creating your own stack script, you exploit the 30000 character limit for your own code. Second, your custom code will not be accessible if you placed it in with the master stack script, then upgrade to a new version of WaveTrak. By collecting your own code into one stack, all you have to do with a new version of WaveTrak is to add a `start using myCustomStackName` instruction in the new master's `openStack` handler.

Trace Card Globals and Wave Descriptors

There are eight important globals intimately associated with trace cards that deserve special mention. These eight globals, called *wave descriptors*, precisely describe a wave in real-world units. If you follow the guidelines outlined in this section, any new buttons you write will be automatically compatible with all of the trace cards' export, plotting, cursor readout and zooming capabilities described in earlier chapters. This will also ensure that any new data you generate can be directly processed by the many analysis buttons in the Button Bank with little or no modification.

Each time you go to a new trace card, the `openCard` handler copies the wave from the background field named 'data' to a global variable called **theWave**. Equally important is that the name of **theWave** (i.e. the string "theWave") is placed in a global called **gList**.

All standard WaveTrak functions that operate on one or more waves expect the name(s) of the global *variables containing these waves to be passed in the global* ***`gList`***. The script of the 'Make Sine' button in the Button Bank is a good example of how to log multiple waves with WaveTrak's standard operations. Here, two sine waves are generated and stored into the globals `w0` and `w1`. By placing their names (i.e. `put "w0,w1" into gList`) as a comma-delimited list into **gList**, you ensure that both waves will be plotted and zoomed together.

The remaining wave descriptors are stored in the following globals, and allow a complete description of a wave. The X values, in real units, of the first and last points, contained in **leftX** and **rightX**, define the range of the acquisition in the horizontal direction, whereas **topY** and **bottomY** define the full-scale vertical range. The first pair of globals allow functions to map a wave's point *numbers* into true time (or any other dimension) values, and the second pair will map the full-scale Y values into real units of measure, such as mV.

Example 1:

You acquire a 1000 point wave at 10 µs/sample. The data are encoded with 12 bits of resolution, as signed integers. You also know that the converter will output the maximum count of 2047 when fed a +10 V input, and the minimum count of -2048 with -10 V. This is how you would define the four globals:
• **leftX** = 0, because the first point of any wave in the time domain is defined as time 0 (Fig. 9-1).
• **rightX** = time corresponding to the last point = sample Interval*(number of points-1) = 10µs*(1000-1) = 9990µs.
• **topY** = 10 V or 10000 mV
• **bottomY** = -10 V or -10000 mV.

In reality, WaveTrak functions don't care much about these four globals; they merely serve to linearly map X,Y values of raw waves (i.e. point numbers and integer counts from the A/D converter, respectively) into units that are more meaningful to the user (i.e. µs and mV, respectively). You are simply telling WaveTrak: "when the cursor is at the very top right corner of the display window, write 9990 µs and +10000 mV as the cursor readout", or in the case of Copy as Y: "when a point in an integer wave has the value -2048, send -10000 mV to the clipboard". Any intermediate X or Y values will be linearly mapped according to these four globals.

Scripting

To complete the description of any signal, you have to define the *units* used for both X and Y values. Example 1 can be modified as follows, without changing the meaning of the data:

Example 2:

• **leftX** = 0.
• **rightX** = 9.990
• **topY** = 10.0
• **bottomY** = -10.0
• **Xunit** = "ms"
• **Yunit** = "V"

Here we defined the time and voltage in milliseconds and volts, adjusting the range of the first four globals accordingly. This is what will be displayed by the cursor readout or exported using Copy as Y. You are free to choose the ranges and units that best suit your data. A good example is in the spectral analysis buttons, where the X and Y units are changed to the frequency domain (Hz and dB). By updating these standard globals, the same plotting and exporting functions will correctly handle the new data.

One final global is used frequently in a wave description: the **baseline**. This value is copied from the Readings field by `openCard` and is used to draw the horizontal baseline in the display window, as well as the value about which trace areas are computed by the 'Trace Area' and other buttons. A wave's baseline is essential if you want to analyze the AC component of a signal that floats on a variable DC level. The baseline is always interpreted in relation to **topY** and **bottomY**.

The advantage of this mechanism of wave description is that once you define these eight globals, you can be sure that WaveTrak's standard functions will plot, export and analyze your waves correctly. For example, the `ReplotWave` handler in the stack script is widely used to redraw the display after a dialog box is dismissed, for instance. If you place the names of 16 globals in **gList** that you want to overlay for example, your plot will be correctly updated from anywhere within the stack until you (or one of WaveTrak's scripts) change the values of these eight globals. One immediate consequence is that all waves logged together in **gList** must have the same number of points, be of the same type, and have the same **leftX**, **rightX**, **topY** and **bottomY** parameters, as well as identical **Xunit** and **Yunit**, or the result

of any overlay will either generate an error or be meaningless.

These guidelines may appear somewhat complicated at first, but after examining a few of the standard scripts in the buttons, and trying out a few of your own, we are sure you will quickly become comfortable with writing scripts which conform to the standard environment. The advantages far outweigh the minimal initial effort.

In Summary

- A wave is a set of points, evenly spaced in time (or other dimension), along with information specifying the data type and the number of points it contains. Elements in a wave can be integers or floating point numbers.

- Wave descriptors (stored in global variables) further describe a wave, supplying information such as vertical and horizontal range, units of measure and baseline.

- XCMDs and XFCNs can only *return* data in global variables. `AcqWave` and related commands expect the *names* of globals where you want your waves returned.

- HyperCard fields can store a maximum of 30000 bytes, which is the upper limit for the size of encoded waves that can be saved in trace cards. Waves manipulated in memory are limited only by RAM (which can be greatly extended with virtual memory).

- Use the example scripts from the Single and Multiple buttons to construct your own acquisition buttons, or expand the scripts in the Import XY and Paste XY buttons to enhance the importing capabilities if you own stack.