WaveTrak Handlers

The commands or functions that WaveTrak executes when it performs its tasks are written as a combination of HyperTalk scripts and XCMDs/XFCNs; the latter are written in C and assembly language for maximum speed. Handlers are chunks of HyperTalk that intercept and *handle* messages, hence the name *handler*. Any segment beginning with `on` ... and ending with `end`... is a handler; the most common one is the `mouseUp` handler present in most buttons. The advantage of writing in HyperTalk is simplicity and the ability to easily modify the instructions. The one disadvantage is execution speed. Some functions such as real-time data acquisition and number-crunching operations like the FFT must be written in C or assembly because of either the time constraints or sheer computational demands. The disadvantage, of course, is that such code modules are more difficult to write and modify. We tried to strike a balance between performance and flexibility by combining the two programming approaches. This chapter describes some of the more important HyperTalk handlers available to you when you write your own functions. Some of the handlers are used internally by WaveTrak and are not documented; you will normally never need to modify these. The next chapter describes in detail all of WaveTrak's XCMDs and XFCNs found in the master, A/D Lib and DSP Lib stacks.

Stack Script

1. `openStack`

   The `openStack` handler executes whenever you open the WaveTrak master stack, either when you first launch HyperCard, or when you come back to the master stack from another, like the WaveTrak summary stack. This handler performs a number of important initializations, such as declaring system-wide globals, checking the hardware, setting up custom WaveTrak menus and so on. The entire WaveTrak environment depends on these initializations, so *you should not delete any lines from the `openStack` handler*. However, you can certainly add additional instructions of your own to further customize your stack. For example, you can add more menus to the menu bar, or more menu items under the existing menus as your requirements evolve. The comments in the handler clearly identify where menus are set up; consult your HyperCard manuals for details on how to create new menus, assign messages

and write handlers to execute the appropriate instructions when the menu item is selected.
Some initializations should only be performed once when the stack is first opened. The **openedOnce** global senses whether the stack had been previously opened.

2. `closeStack`

   This handler resets the menu bar to the standard HyperCard menus when you leave WaveTrak. Add any other commands you need here.

3. `UpdateSysParams`

   This handler copies default values from various fields (such as in the System Parameters card) into global variables. If you add your own default fields on existing or additional cards, place instructions to copy these fields into your own globals in this handler to ensure that values will be available throughout your stack.

4. `ErrNum`

   `ErrNum` displays a dialog with a brief message describing the most recent error, if any, generated by one of the WaveTrak XCMDs or XFCNs. All XCMD errors are returned in the global **XCMDErr**. If **XCMDErr** = 0 then no error occurred and `ErrNum` does nothing. The error messages are stored in the ErrorList card along with an error number. When you call `ErrNum`, you pass the error *number* as the only parameter, and `ErrNum` looks up the appropriate message from the list in the ErrorList card. You can look up an error by its number in the chapter on WaveTrak errors for more information. It's a good idea to call `ErrNum` after every XCMD to make sure that it completed successfully. Otherwise you will only hear a beep and not know what caused it.

Example 1:

Here's how to report errors generated by XCMDs (`AcqWave` as an example).

```
  global XCMDErr, theWave
  . . .
  AcqWave sampleInterval, npoints, startMUX,
endMUX,"theWave"
  ErrNum XCMDErr  -- pass the error number to ErrNum
```

If something went wrong during `AcqWave`, calling `ErrNum` immediately afterwards will display an error dialog with a message. Otherwise, `AcqWave` sets **XCMDErr** to zero and calling `ErrNum` does nothing. Examine the buttons in the button bank for more examples of how to handle WaveTrak errors.

If you want to display a certain error dialog then look up its number in the chapter on WaveTrak errors and call `ErrNum` directly with that number:

Example 2:

Here's how to report a specific error, for example, informing the user that a certain function is not implemented.

```
 -- 1 is the error code for 'Function not implemented
error'
 ErrNum 1
```

5. `doMenu`

Some menu selections have to be handled differently depending on the context. For example 'Delete Card' must update different index fields depending on whether you choose this item from a trace or root card. The `doMenu` handler intercepts menu selections and re-directs those that need special attention.

6. `newRoot`

This handler creates a new root card, ensuring that it is linked to the rest of the stack in the appropriate order, that the date stamp is entered, and that default fields are initialized.

7. `PlotData`

`PlotData` is a general purpose handler that plots a table of X,Y data on the current card. The parameters are as follows, in order:
• `theData`: comma-delimited X,Y data pairs.
• `Xmin, Xmax`: the minimum and maximum X values in `theData`.
• `Ymin, Ymax`: the minimum and maximum Y values in `theData`.
• `gridDots`: TRUE for dots to be plotted in a grid over the data.

The 'Plot Abs Areas' analysis button in the root cards uses this handler to display its results. Press this button for an example.

6. `translateToReal`

This is a *function* (i.e. it returns a result) which converts a binary integer into a real value in the context of a pre-defined full scale range and data type (see the scripting chapter for details on WaveTrak data types) used to encode the wave. You will use this and the related function `translateToBinary` extensively to convert data from the A/D converter into real values.
The parameters are as follows, in order:
• `i`: the binary integer.
• `minFS, maxFS`: the minimum and maximum real full-scale values.
• `waveType`: the binary encoding scheme (e.g. -12, +16, etc.).

Example:

You have just read a single integer value (`i`) from the A/D converter (for example +456) and want to know the real voltage that this value represents. You look up the full-scale range for the A/D channel just sampled from the External A/D gain table in the System Parameters card (or from the **FSTable** global) and find that it's -10000 to +10000 mV. You also know (from the **ADCbits** global) that your converter is configured for 12 bits, signed two's complement encoding, so the data type is -12:

```
  put translateToReal (456,-10000,10000,-12) && "mV"
```

will write '2229.54823 mV' in the message box, which means that a converter code of +456 represents a real voltage value of about 2229 mV.

```
  put translateToReal (456,-10,10,-12) && "V"
```

would be just as correct, except the result would be reported in volts rather than mV.

See the 'Show Single Mean' button in the Button Bank for another example.

7. `translateToBinary`

This function does the opposite of `translateToReal`; it converts a real value into a binary integer in the context of a pre-defined full scale range and data type. If the binary value would have been beyond the range allowed by the `waveType` parameter, `translateToBinary` returns 100000 to signal the error condition (100000 was chosen because it exceeds the largest 16 bit integer).
The parameters are as follows, in order:
• `r`: the real value
• `minFS, maxFS`: the minimum and maximum real full-scale values
• `waveType`: the binary encoding scheme (e.g. -12, +16, etc.)

Example 1:

You want to acquire a wave when it crosses a threshold of -0.789 V, using the `AcqWaveThresh` XCMD (see the following chapter for details). This XCMD requires a binary value as the trigger (threshold) level. You look up the full-scale range for the required A/D channel from the External A/D gain table in the System Parameters card (or from the **FSTable** global) and find that it's -1000 to +1000 mV. You also know (from the **ADCbits** global) that your converter is configured for 12 bits, signed two's complement coding, so the data type is -12:

```
put translateToBinary(-0.789,-1,1,-12) into thresh
AcqWaveThresh sInt,npoints,sMUX,endMUX,¬
thresh,slope,timeout,"theWave"
```

Note that you must be consistent in passing volts or mV, but not a combination. `thresh` will get the value -1616, and `AcqWaveThresh` will correctly trigger at -0.789 V.

Example 2:

You now want to repeat the acquisition, but with a threshold of 1.23 V:

```
put translateToBinary(1.23,-1,1,-12) into thresh
AcqWaveThresh sInt,npoints,sMUX,endMUX,thresh,slope,¬
timeout,"theWave"
```

This time, `AcqWaveThresh` reports a 'Threshold out of range error'. 1.23 volts was beyond the ± 1 V full-scale so `translateToBinary` purposely returned a large integer (100000) to signal this error. We neglected to check if `translateToBinary` returned a valid binary number.

See the 'Single Thresh' button in the Button Bank for another example.

8. `getWaveType`

This function returns the data type of the wave passed as the first (and only) parameter. Note that unlike XCMDs, the wave is passed by *value* so its name must not be enclosed in quotes. See the scripting chapter for details on standard WaveTrak data types.

Example:

```
  global theWave
  ...
  AcqWave sampleInterval, npoints, startMUX,
endMUX,"theWave"
  put getWaveType (theWave)
```

Assuming that your converter is configured for 12-bit signed two's complement coding, the last line will write '-12' in the message window. Note that `theWave` is passed by *reference* (i.e. its *name* is passed, in double quotes) to `AcqWave`, but is passed by *value* (no quotes) to `getWaveType`. As a result, waves passed to `getWaveType` need not be stored in global variables.

9. `ReplotWave`

   This handler will redraw (using the `DrawWaveCoords` XCMD) in the display window, any wave(s) whose global names are stored as a comma-delimited list in the **gList** global. `ReplotWave` will only redraw the waves in a trace card or the Scope card. Therefore, you can call `ReplotWave` any time (after dismissing a dialog box for example) from any card without fear of having waves drawn in wrong card. You have to set up the following globals for `ReplotWave` to work correctly: **gList**, **dispRect**, **leftX**, **rightX**, **topY**, **bottomY**, **baseline**, **Xunit**, **Yunit**. The chapter on scripting explains what these globals mean and how to correctly initialize them.

Root Background Script

1. `openBackground`, `closeBackground`

   These two handlers update menus whenever you jump into and out of the root background. You can add your own menu operations here as well.

2. `Single`

Most of the trace acquisition commands are initiated using buttons in the trace cards. However, there must be a way to create the first trace card under a new root. This handler responds to the 'Single A/D' menu item under the 'Acq' menu by sending a mouseUp message to the 'Single A/D' button in the trace card.

3. `deleteRoot`

   This handler deletes the current root card as well as all traces belonging to it. It also ensures that the remaining roots are re-linked and re-numbered correctly, and that the Root Titles field in the Home Card is updated. `RenumberRoots` and `UpdateTitlesField` handlers help in this task.

Trace Background Script

1. `openBackground, closeBackground`

   These two handlers update menus whenever you jump into and out of the trace background. You can add your own menu operations here as well.

2. `openCard`

   This important handler copies vital information into a series of globals whenever you go to a new trace card, then draws the previously saved wave on the screen. To avoid duplicating information and increasing the size of the stack, digitized data are stored in compressed format in a hidden field named 'data', and not as a graphic on the trace card. Thanks to several highly optimized XCMDs, WaveTrak is able to decompress the data from the field and draw the wave almost instantly each time you open a new trace card. Let's go over in detail some of the more important instructions in the `openCard` handler. Below is a listing of the handler, with line numbers added so the following discussion can clearly refer to specific parts of the code:

```
1 on openCard
2  global rootId,XCMDErr,theWave,HParams,Readings
3  global
gList,leftX,rightX,topY,bottomY,baseline,Xunit,Yunit

 -- copy wave & info into globals for other handlers
4  put bg fld "data" into theWave
5  put "theWave" into gList  -- name of wave(s) to draw
6  put bg fld "HParams" into HParams
7  put bg fld "Readings" into Readings

 -- if it's a wave, draw it
8  if line 1 in HParams = 1 then  -- is it a wave?
9    put 0 into leftX
10   get line 2 in HParams
11   put (it-1) * (line 3 in HParams) into rightX
12   put "µs" into Xunit

13   get line 4 in HParams  -- full scale and units
14   put item 1 in it into bottomY
15   put item 2 in it into topY
16   put item 3 in it into Yunit
17   put line 2 in Readings into baseline

18   ReplotWave
19 end if

...

end openCard
```

Line 1 introduces the `openCard` handler, and lines 2 and 3 declare globals that will be initialized and made available to other handlers while you are at this trace card. Several of these globals provide a complete description of the wave so other handlers can draw it, analyze it or export it. See the section on wave descriptors in the scripting chapter for more details. The actual compressed wave is copied into the global **theWave** in line 4 and the name of this global is put into **gList**. You will recall that WaveTrak XCMDs 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**. Since we have a single wave per trace, only a single name is copied into **gList**. Data in background fields 'HParams' and 'Readings' are copied into variables for quicker access in lines 6 and 7.

Although the current version of WaveTrak holds only waves in the 'data' fields, this may change. The data identifier is always stored in line 1 of the HParams pop-up field and is set to 1 for a wave. Line 8 checks if the trace really contains a wave before proceeding to draw it. The previous chapter explained the eight wave descriptors that must be initialized for any wave to be processed correctly by WaveTrak. They are as follows:

leftX: the time of the first point in a wave, defined as zero (line 9).

rightX: the time of the last point in a wave, computed as the number of points in the wave (line 10) minus 1 (see Fig. 9-1) multiplied by the sampling interval (line 11). The sampling interval is stored in line 3 in the HParams field.

Xunit: unit of measure for the X-axis, defaults to 'µs' for waves digitized in the time domain (line 12).

bottomY, **topY**, **Yunit**: the minimum and maximum full-scale values at the time of the acquisition as well as the unit of measure for the Y-axis are stored as three items in line 4 in the HParams field. Lines 13 to 16 in the script copy this information to their respective globals.

baseline: the DC level of the signal may have been stored in line 2 of the Readings field (otherwise it defaults to zero); line 17 copies this value to the corresponding global.

Along with **gList**, these globals provide a complete description of our wave. Line 18 can then call the standard `ReplotWave` handler which will know exactly how to draw the wave, zoom it, and display the correct cursor readout and units, using the above globals. You can see how straightforward it would be to modify this script to handle any other data. For instance, if you wanted to store only frequency spectra on the trace cards, you would put "Hz" into **Xunit** and "dB" into **Yunit**.

The remainder of the `openStack` handler does some routine housekeeping.

3. `SaveIgorTEXT`

Although this handler simply responds to the 'Save as Igor Text' menu selection, it deserves brief mention because of the potential it offers for creating automated Igor macros. The designers of Igor endowed their application with the capability of not only importing data from a disk file, but also executing any number of standard Igor commands that may be appended to the end of the data. You will need to refer to the Igor manual to learn how to do this. There is a comment line in the `SaveIgorText` handler that tells you where to add your own Igor commands to the footer that is written after the data. Compose any valid Igor command line and add this text as follows:

```
   put "any valid Igor command line" & return after
footer
```

You can string together as many commands as you wish. When you read this file into Igor, the data will be imported and any valid commands will be executed. In this way, you can instruct Igor to perform repetitive tasks. If you need Igor to do many different functions, or the macro you need is long, you could create a new card with several fields, each containing a collection of Igor commands. You can select the appropriate field and append its contents to the footer depending on what you need to accomplish. Most wave processing and analysis can be more easily done using WaveTrak's powerful XCMDs. For page layout and graphing, however, Igor is an excellent choice.

4. `newTrace`

The `newTrace` handler is analogous to the `newRoot` handler in the root background. It is used to create and link new trace cards to the end of the series of traces under the current root. Default HParams and Readings fields are filled in, as well as the time stamp, marks, and so on. You can add your own functions here as well.

5. `deleteTrace`, `RenumberTraces`

These two handlers ensure that any remaining traces will be correctly linked and numbered when you delete a trace card.

There are many more scripts in fields, buttons and card scattered throughout the WaveTrak stack. The remaining handlers are used internally and you normally will never need to worry about them. However, we encourage the more adventurous users to explore these scripts to learn how WaveTrak works. *You should modify any undocumented handlers with caution as this may interfere with WaveTrak's normal operation*. It's always a good idea to keep a backup copy of the original WaveTrak disk close by, and more importantly, of your most recent working modification as well.

Tip:

The HyperCard environment is very forgiving, and you will usually end up with an error message (from HyperCard or WaveTrak) if something goes wrong. However, if a script is interrupted by an error, the stack might be left in an inconsistent state, and you may find yourself unable to navigate. The Home Card was designed to 're-orient' all the directions and menus, so jump to the Home Card when you run into trouble, by selecting 'Home Card' from the Go menu, or typing command-1 (remember to fix the problem script though!).

In Summary

- WaveTrak commands and functions consist of a combination of HyperTalk scripts for maximum flexibility, and XCMDs/XFCNs for maximum speed.
- Commands implemented as HyperTalk scripts are called handlers and respond to messages sent by events (such as mouse clicks) or by other handlers.
- Most of WaveTrak's housekeeping functions are written in HyperTalk and are customizable by the user.