SSN Jeremy John Ahouse 457-43-9290
SSN Eric Carlson 574-54-6965
Tel. # 415-644-9842

We are biologists who have found ourselves doing ever more computer work and lots of HyperCard scripting. Jeremy wrote a chapter in the new Howard Sams intermediate scripting book Tricks of the HyperTalk Masters and Eric is employed by Apple Computer Inc. as a multimedia software engineer.

Hyper Etiquette

XCMD Etiquette

© 1989 Jeremy John Ahouse & Eric Carlson
Waves Consulting & Development
1228 Carleton
Berkeley, CA
(415) 644-9842
AppleLink CYNIC

This note suggests several approaches to standardizing the interaction of externals with HyperTalk in a user-oriented way.

We have noticed that as XFCN's and XCMD's are promulgated many have become difficult to use, and are often difficult to interpret in the context of reading a script. There is no reason that externals should not retain the spirit of HyperTalk. We will make several recommendations to this end and then offer example code that illustrates our thoughts. (note: We will refer to both XCMD's and XFCN's as XCMD's.)

HyperCard has given many people a chance to use their computers in ways that were until recently restricted to "programmers." The distinction between users and programmers has been eroded by a new class of user/programmer called scripters. We'll give away the moral of this story now; when writing XCMDs you should treat scripters the way you would treat users if you were writing traditional Macintosh applications.

There are really two issues that we need to address. The first is making an external easy to use, the second is making scripts that use externals easy

to read and understand. These two are not mutually exclusive.

We begin by listing four problems and then follow with discussions and possible solutions for them. We will end by illustrating our points with source for a StringLength XFCN.

The Problems

1) Many externals obscure the flow of HyperTalk scripts. This makes them harder to understand (and debug).

2) When an error occurs during the execution of an external, how should this be reported to the user/scripter?

3) A user forgets the parameters of your external and there isn't a standard way to find out what they are.

4) A user doesn't want to include a long list of parameters if only one feature of an external is used.

Solutions

1) Making scripts read well requires XCMD's that are well named and parameters that are easy to understand. We illustrate this point with a counter example:

```
put xseven(2, no, 4, h, 1) into msg
```

Try to use words for input parameters whenever you can. Obviously in some cases it will be much clearer to pass numbers. A rule of thumb is: use numbers only if you are actually working with a number in the external, like the number of items in a list, or lines in a container. Finally numbers are appropriate if the parameters in the XCMD can get their values from HyperTalk functions (like `max`) or properties (like `textHeight`) that return numbers.

A way to avoid naming your XCMD obscurely is to not ask too much of it. Allow your external to do a reasonable number of things well. If you have lots of great ideas write more than one XCMD. Remember that XCMD's *extend* HyperTalk.

Another aspect of naming externals is to try to make them read well. This is particularly important for XFCNs, which may become part of HyperTalk statements. Try this test. How does your function sound/read in the following contexts:

```
get myFunction()
put myFunction() into msg
put item 4 of myFunction() into temp
```

Names that start with verbs don't work well. XCMDs, on the other hand, often read well if they start with verbs.

2) Reporting errors is always a problem. There are many levels of users and while some will want to handle error codes themselves, others will benefit from a less subtle solution. Make the last (optional) parameter either "Dialog" or "noDialog" with the former as default. Here is an example;

```
functionThatDanLeftOut(param1, param2, "Dialog")
```

or, equivalently:

```
functionThatDanLeftOut(param1, param2)
```

In these cases the external will return error codes in a dialog and in the result, whereas

```
functionThatDanLeftOut(param1, param2,
   "noDialog")
```

will report return error conditions in the result only. This convention will allow users to suppress error messages that stop the flow of a script and to handle the error conditions on their own if they so choose.

It should also be apparent from the tone of this note that we don't encourage the idea of returning errors like this;

```
-202
```

rather do this:

```
"The Mac seems to have chewing gum in the
   speaker."
```

It seems that this recommendation may be difficult for people who implement whole systems that reside outside of HyperCard and who use a set of externals to communicate with their extra-HC system. We are thinking here of search engines, databases, etc… For those who feel strongly about the need to return error conditions numerically we suggest offering your users a function which returns a description of an error condition when passed the error number.

```
put Error("-202") into msg box
```

would put

```
"The Mac seems to have chewing gum in the speaker."
```

into the msg box. The point here is to make interactions with externals as easy to use as possible.

3) XCMDs are often documented only within the simple stacks written to distribute and demonstrate them. It is inconvenient for a scripter to have to find and open that stack if they forget the syntax for an external during stack development. Additionally, as newer (debugged!) versions of externals come out it is often difficult to know which version of an XCMD is in a stack. Support the following forms for your external;

```
functionThatDanLeftOut("?")
```

should reports back the syntax for the external without performing its function, ie.;

```
functionThatDanLeftOut("param1", "param2", "param3")
```

would be returned by the XFCN (or XCMD).

If some of the parameters are optional surround them with the <> symbols ie.;.

```
commandThatDanLeftOut("param1", <"param2">, <"param3">).
```

Version and copyright information can be made available to scripters in the same way:

```
functionThatDanLeftOut("??")
```

or

```
commandThatDanLeftOut "??"
```

should return the copyright information and the version for the external. As first written, this article reccomended using the copyright symbol ("©") for version and copyright information. Upon review Fred Stauder noted that this symbol is not available on all international keyboards, and so reccomended a change. Thanks Fred!

A pair of simple Pascal functions to check for and respond to these requests might look like this :

```
procedure reportToUser (paramPtr: XCmdPtr;
    msgStr: str255);
{}
{ report something back to the user.  we always fill }
{ in the result field of the paramBlock, and optionally }
{ use HC's "answer" dialog unless requested not to }
{}
 var
  tempName: str255;
begin
 paramPtr^.returnValue := PasToZero(paramPtr, msgStr);
{check the last param to see if the user requested that }
{ we suppress the error dialog }
 ZeroToPas(paramPtr,
paramPtr^.params[paramPtr^.paramCount]^, tempName);
 UprString(tempName, true);
 if tempName <> 'NODIALOG' then
  SendCardMessage(paramPtr,

                                                            concat('answer "',
msgStr, '"'));
end;      { procedure }

function askedForHelp (paramPtr: XCmdPtr;
    syntaxMsg: Str255;
    copyRightMsg: Str255): boolean;
{}
{ check to see if the user sent a '?' or a '??' as }
{ the only parameter. if so we will respond with }
{ the calling syntax or the copyright/version info }
{ for this external }
{}
 var
  firstStr: str255;
begin
 askedForHelp := false;
 if paramPtr^.paramCount = 1 then
  begin
   ZeroToPas(paramPtr, paramPtr^.params[1]^, firstStr);
                                                            { what is the first
param? }
   if firstStr = '?' then
    begin
      reportToUser(paramPtr, syntaxMsg);
      askedForHelp := true
    end   { asked for help }
   else if firstStr = '??' then
    begin
      reportToUser(paramPtr, copyRightMsg);
    askedForHelp := true
   end;  { asked for copyright info }
  end;   { one parameter passed }
end;     { function }
```

Many externals (wise externals?) check the parameterCount and return some of this information if the number of passed parameters is wrong. Most of these will continue to function properly if a user presents the external with a "?" or a "??", but the point is to make this method standard so that users know to use it. Adopting this approach will give scripters a standard way to query XCMDS, and will give us a way to internally document externals.

4) Support default values for your externals. This means that a user is required to pass only those paramameters that are necessary. In the StringWidth function that we discuss below, if only a string is passed the function defaults to the HyperCard default text size, font and style - 12 point, geneva, plain. This approach seems to offer a good combination of flexibility and clean HyperTalk. If taken to the extreme this approach can also make it very difficult to elucidate the purpose an XCMD when reading through a script, so keep point 1 in mind as you decide on optional parameters and default values.

Problems?

Not all of these recommendations will be universally applicable. Doubtless someone has written an external which must be passed "?" or "??", but try to remember the spirit of these approaches. Make the external easy to use, flexible, easy to read (for debugging if not aesthetics), and finally treat external users like Macintosh users. HyperCard has made "~~programming~~" (whoops) "scripting" available to many people who never thought they would ever have so much control over their computer. It is vital that we do what we can suppress the tendency for the techno-macho/techno-less macho dichotomy to take hold (or should we say widen).

An Example

What follows is the source for an XFCN written in Think's Pascal which tries to follow some of our own advice. This XFCN returns the width in pixels of a string passed to it. It is similar in function to Fred Stauder's XCMD from the March issue of MacTutor, but we have given it some additional functionality as well as writing it as an XFCN (it is a function after all). Fred's implementation contained no information about the font, style, and size of the text. This can be a fatal flaw in many cases. The function we present allows you to specify all of these attributes. It is called as follows:

```
stringWidth (container, font, size, style,
    <noDialog>)
```

Finally, here is a description of what our example code does:

StringWidth first checks the parameter block pointer to see if any parameters were passed (although most of the parameters have default values, it is fairly difficult to guess what string the user wishes to use). Assuming the user is somewhat confused about the XFCN's use if no parameter are passed, we send back the calling syntax.

Next we check to see if they have explicitly asked for the calling syntax or for copyright/version information, and respond appropriately if so.

Once we have the string to measure, we need to determine what the font, size and style parameters are, as they can make a huge difference in the string's width. HyperCard's default font is geneva, so if the user doesn't pass any information about the font we use it as our default too. HyperCard displays a button or field in geneva if the font which was originally assigned to it is not available, but the textFont property for that field or button returns the number of *the original font.* Thus we must check to see if a number is passed as the font parameter, and use geneva when we find one. The final check on the font parameter is to make certain that the name passed is available. If the font name is misspelled or not available in an open resource file, the toolbox call GetFNum returns 0. Because this is also the correct font number for Chicago we call GetFontName and compare the name returned with the name passed as a parameter to see if the requested font is available. In the event of an error we fill `the result`, and if the user did not pass "noDialog" as the last parameter we also report the error via HyperCard's `answer` dialog.

The third parameter is the font point size. If none is passed we use HyperCard's default, 12 point.

The fourth parameter is the font style. We check this parameter by a simple, if somewhat tedious, series of tests for the presence of each of the possible style options.

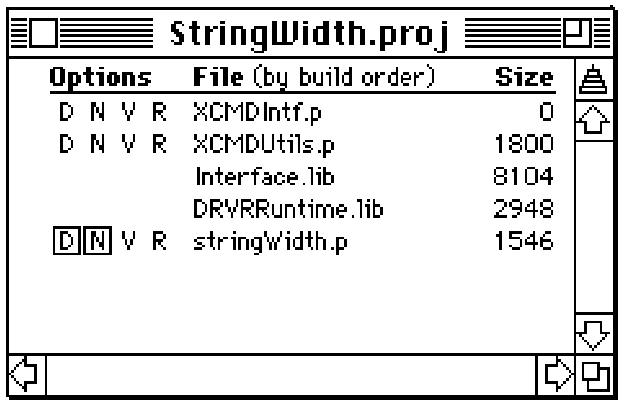
Once we have finally determined all of the parameters, our task is quite simple: set the port to the specified font characteristics, call StringWidth to find the pixel width of the string parameter, and *reset the port back to its original characteristics.* This last step is a small one but it should not be overlooked.

And So…

Scripters who have "cut their programming teeth" on HyperTalk are accustomed to (and perhaps rely upon) HyperTalk's conventions, including code which reads easily and clearly, understandable error messages, and so forth. Remembering that these people are potential users of our externals should help us to write externals in such a way that they extend HyperCard's functionality without departing from its spirit. The distinctions between different kinds of computer users are finally becoming more and more difficult to define, lets do our part to continue the trend.

We hope that these recommendations prove useful.

Good Luck and Good Scripting,
Eric Carlson and Jeremy Ahouse



Options	File (by build order)	Size
D N V R	XCMDIntf.p	0
D N V R	XCMDUtils.p	1800
	Interface.lib	8104
	DRVRRuntime.lib	2948
$\boxed{D}$ $\boxed{N}$ V R	stringWidth.p	1546

Figure 1. The project window for the example presented below. Note that because we compile to a code resource we must use DRVRRuntime.lib library rather than Runtime.lib (the later references its globals through register A5, a definite no-no for an XCMD).

```
unit stringWidthUnit;
{}
```

```
{ LSP Project contains: }
{ XCMDIntf.p }
{ XCMDUtils.p }
{ Interface.lib }
{ DRVRRuntime.lib }
{ stringWidth.p (this file ) }
{}
{ syntax is:stringWidth(stringHolder, font, size,}
{ style,<noDialog>) }
{ the parameters should be specified as hypercard }
{ reports them, ie. }
{ stringWidth("this is a dummy string", "PALATINO",}
{ "14", "BOLD,ITALIC", "noDialog") }
{}
{ copyright (©)  Eric Carlson and Jeremy Ahouse }
{ April 29, 1989 }
{ Waves Cosulting and Development }
{ Berkeley, CA    94792 }
{ free for non-commercial use only }
{}
interface
uses
 XCMDIntf, XCMDUtils;

procedure main (paramPtr: XCmdPtr);

implementation

{-----------------------------------------------}

procedure reportToUser (paramPtr: XCmdPtr;
    msgStr: str255);
{}
{ report something back to the user.  we always fill }
{ in the result field of the paramBlock, and optionally }
{ use HC's "answer" dialog unless requested not to }
{}
 var
  tempName: str255;
begin
 paramPtr^.returnValue := PasToZero(paramPtr, msgStr);
{check the last param to see if the user requested that }
{ we suppress the error dialog }
 ZeroToPas(paramPtr,
paramPtr^.params[paramPtr^.paramCount]^, tempName);
 UprString(tempName, true);
 if tempName <> 'NODIALOG' then
  SendCardMessage(paramPtr,

                                                              concat('answer "',
msgStr, '"'));
end;     { procedure }

function askedForHelp (paramPtr: XCmdPtr;
    syntaxMsg: Str255;
    copyRightMsg: Str255): boolean;
{}
{ check to see if the user sent a '?' or a '??' as }
{ the only parameter. if so we will respond with }
{ the calling syntax or the copyright/version info }
{ for this external }
{}
 var
  firstStr: str255;
begin
 askedForHelp := false;
 if paramPtr^.paramCount = 1 then
  begin
   ZeroToPas(paramPtr, paramPtr^.params[1]^, firstStr);
                                                              { what is the first
param? }
   if firstStr = '?' then
    begin
      reportToUser(paramPtr, syntaxMsg);
      askedForHelp := true
    end   { asked for help }
   else if firstStr = '??' then
    begin
      reportToUser(paramPtr, copyRightMsg);
      askedForHelp := true
    end;  { asked for copyright info }
  end;    { one parameter passed }
end;      { function }

procedure widthOfString (paramPtr: XCmdPtr);
{}
{ set the specified pen characteristics and get the }
{ width of the string with the toolbox routine }
{ StringWidth }
{}
 label
  1;
 var
  passedString, errorStr, tempName: str255;
  copyRtStr, syntaxStr: str255;
  oldFont, oldSize, fNum, fSize, width: integer;
  fName, sizeString, theStyleStr: Str255;
  oldStyle, theStyle: Style;
  HCPort: GrafPtr;
begin
 syntaxStr := 'stringWidth(stringHolder, font, size, style, <"noDialog">)';
 copyRtStr := 'v1.0, ©1989 Waves Consulting and Development, Berkeley
CA.';
 if paramPtr^.paramCount = 0 then
  begin
                           { no parameters passed, report our calling syntax }
   reportToUser(paramPtr, syntaxStr);
  goto 1;
 end;

if not (askedForHelp(paramPtr, syntaxStr,
                                                   copyRtStr)) then
 begin
  GetPort(HCPort);                                                  { grab the
port }
  with HCPort^ do
   begin
   oldFont := txFont;                       { save current typeface }
   oldSize := txSize;                       { save current size }
   oldStyle := txFace;             { save current style }
   end;

  ZeroToPas(paramPtr, paramPtr^.params[1]^,

            passedString);{ get the string to trim }

            { do we have a font name parameter? }
  if paramPtr^.paramCount > 1 then
   ZeroToPas(paramPtr, paramPtr^.params[2]^,

                      fName)
                                                                    { which
font? }
  else
   fName := 'GENEVA';
                                                                    { no font
passed, use HCs default }

  fNum := StrToNum(paramPtr, fName);
{ check to see if a number was passed as the font }
{'name' parameter. if so, we assume that the font }
{ which HC wants to use for the field/button is not }
{ available in the current system. in this case geneva }
{ is being used instead, so we should use it too! }
  if fNum <> 0 then
   fName := 'GENEVA';
  GetFNum(fName, fNum);                      { get the font number }
{ if we call for an unavailable font (not present in }
{ this system, name spelled incorrectly, etc, GetFNum }
{ returns 0, which also happens to be the correct }
{ number for CHICAGO.  thus we now check to see if }
{ the name for the font num is the same as the font }
{ name passed to us, or if our user is requesting the }
{ impossible }
  GetFontName(fNum, tempName);
  UprString(fName, true);
  UprString(tempName, true);
  if tempName <> fName then
   begin
    errorStr := concat('Sorry, the font ', chr(39),

                              fName, chr(39),' is not avaliable.');
    reportToUser (paramPtr, errorStr);
    goto 1;
   end;

  if paramPtr^.paramCount > 2 then
```

```
                                                                    { do we
have a size parameter? }
   ZeroToPas(paramPtr, paramPtr^.params[3]^,

            sizeString) { font size in string form }
  else
   sizeString := '12';

            { no size passed, use HCs default }
  fSize := StrToNum(paramPtr, sizeString);

            { actual size }

  theStyle := [];
                                                                     { is
there a style parameter? }
  if paramPtr^.paramCount > 3 then
   begin
             ZeroToPas(paramPtr, paramPtr^.params[4]^,

                                   theStyleStr);         { which style(s)? }
              UprString(theStyleStr, true);
                                                     { convert to uppercase }

                                             if pos('BOLD', theStyleStr) > 0 then
                        theStyle := theStyle + [bold];
                                             if pos('ITALIC', theStyleStr) > 0 then
                        theStyle := theStyle + [italic];
                                             if pos('UNDERLINE', theStyleStr) > 0
then
                        theStyle := theStyle + [underline];
                                             if pos('OUTLINE', theStyleStr) > 0
then
                        theStyle := theStyle + [outline];
                                             if pos('SHADOW', theStyleStr) > 0
then
                        theStyle := theStyle + [shadow];
                                             if pos('CONDENSE', theStyleStr) > 0
then
                        theStyle := theStyle + [condense];
                                             if pos('EXTEND', theStyleStr) > 0 then
                        theStyle := theStyle + [extend];
   end;

                        { now setup the port with the specified font }
                        { attributes }
  TextFont(fNum);                            { set it to the current font, }
  TextSize(fSize);                           { and the size, }
  TextFace(theStyle);             { and the style... }

  width := StringWidth(passedString);

                                                                    { how
wide is that string? }

                        { we mustn't forget to clean up after ourselves, }
                        { reset HC's port to the entry conditions }
  TextFont(oldFont);              { reset the  font… }
  TextSize(oldSize);              { and the size… }
  TextFace(oldStyle);{ and the style }

                        { send back the width }
  paramPtr^.returnValue := PasToZero(paramPtr,


            NumToStr(paramPtr, width));
```

```
end;
```

```
1:                              {bail out point if we run into trouble }
end;

procedure main;
begin
 widthOfString(paramPtr);
end;
end.
```