Mac Programming

Understanding LightspeedC:

Talking Back to HyperCard

by Paul Potts

In my last article (Dec. '88) I described how to create a simple HyperCard XCMD and examined the first half of Hypercard's data structure, the XCmdBlock. Now we will examine the second half of the XCmdBlock and how to call back into HyperCard to use some of its powerful utility routines.

Background

HyperCard stores all variables, even numbers, as strings. Pascal expects that this string will consist of a length byte followed by up to 255 characters. C treats strings slightly differently: in C, a string consists of an arbitrary number of data bytes terminated by a zero byte.

Here is how that a Lightspeed C defines the Str255 data type:

```
typedef unsigned char
Str255[256];
typedef unsigned char *
StringPtr,** StringHandle;
```

The HyperCard XCMD interface code defines a shorter internal format to hold strings:

```
typedef struct Str31
{    char guts[32];
} Str31, *Str31Ptr,
**Str31Handle;
```

Writing XCMDs will be much simpler if you keep in mind that any array can hold either a C-type or Pascal-type string. When calling back into HyperCard it is important to pass it arguments in the proper format. XCMD callbacks necessitate converting back and forth between string types quite often. Your code must remember which type of string lurks in each array: if you pass a string of the wrong type, the results will not be what you expect.

Let us now discuss the second half of the XCmdBlock and how it can be used to call back into Hypercard. Here is the definition:

```
typedef struct XCmdBlock
{
/* ...we will ignore the
first half of the data
structure... */
     char   *entryPoint; /*
to call back to HyperCard */
     short  request;
     short  result;
     long          inArgs[8];
     long
outArgs[4];
} XCmdBlock, *XCmdBlockPtr;
```

In order to call back into HyperCard, your XCMD must jump to the address contained in **entryPoint**. The interfaces written in MPW C suggest this can be done with the following:

```
((ProcPtr)(paramPtr-
>entryPoint))();
```

However, this doesn't work in Lightspeed C. I tried a large number of possible calls in C and assembly language, to no avail. Symantec told me over the phone that they would send me their own interface routines on a disk for ten dollars. By pleading poverty I convinced the representative to tell me how to call back into HyperCard, but I was unable to get his method to work either. In desperation I read the manual, and found a built-in function to call Pascal routines:

```
CallPascal(paramPtr-
>entryPoint);
```

This may not be the best way, but it works and it is simple to read.

The next field in the XCmdBlock, **request**, tells HyperCard just which internal routine you wish to execute (there are twenty-nine of them in the current release of HyperCard). **inArgs** and **outArgs** contain handles to the arguments sent back and forth to Hypercard. Fortunately, you don't need to remember which **inArgs** and **outArgs** go where: Dan Winkler kindly wrote a set of glue routines for MPW C. To use them in Lightspeed™ C, change every reference from the MPW **extended** type to **double**. (I haven't used these routines to see if they work, but this fix will at least get it by the compiler for now). Remember to replace the jumps back into HyperCard with **CallPascal (paramPtr-> entryPoint);**. I have included an abbreviated set of glue routines which are sufficient for building this project, and quite a bit more besides.

The Project

Drawing in Hypercard is slow and painful. Wouldn't it be nice if HyperCard could open up its own windows and use Quickdraw to dynamically draw into them, even using color on a Macintosh II if desired? In my last article the XCMD opened a PICT resource and displayed it on the screen, then went into an event loop and waited for a mouse click or keypress before continuing. I wanted to create a window that was a bit more permanent.

To build my XCMD project, create a folder with the four source code files in it: **working.xcmd.h, working.glue.c, graphPack.c,** and **XCMD.shell.c.** Compile graphPack and the XCMD.shell in a project with MacTraps. (Do not add the other two source code files to the project: they are linked in using the #include directive). Build the code resource (make sure you assign it a unique ID number) and paste it into a HyperCard stack. Define a global container in the stack script, with a line like **global myWindowPtr,** and create a new STR (string) resource in your stack with an ID number of 100. You will also need to add a PICT resource to define the size and content of the initial window.

Since the XCMD's window does not belong to HyperCard, HyperCard doesn't even know it exists. Your stacks and scripts will run behind it, even flipping from card to card, and it will stay there until you explicitly dispose of it. Unfortunately, if you drag the message box window or a desk accessory window in front of it, you will erase part of the content region; an XCMD cannot accept update events, since it does not run in the background like a desk accessory. It is up to the calling stack to make sure the user does not drag the foreign windows around.

Although HyperCard doesn't know about my window, my XCMD

Mac Review

MacInker

Ribbons for 5 Cents

By Lee Cabana

I love the results that a brand new ribbon produces in my ImageWriter II - dark and sharp, especially in the Near Letter Quality mode. Unfortunately, that quality starts to fade after the ribbon has made a few cycles through the cartridge. In the past, I've made do, buying new ribbons when quality was foremost, and setting the once or twice used cartridges aside (wrapped in plastic, and stored in the refrigerator) for future use. I've also used a technique I learned at my first meeting of the Erie Apple Cruncher Users Group—open the cartridge, spray WD-40 over the ribbon, and let the reassembled cartridge sit for a couple days while the ink in the unused portions permeates the entire ribbon. Not a bad technique (the price is certainly right!), but there is only so much ink in the cartridge, and the second spraying doesn't rejuvenate the ribbon as well as the first application.

Some time ago, I noticed advertisements in the Apple user magazines for a product called MacInker, a system that claimed to re-ink ribbons to like new quality. The headline claimed, "Re-ink ribbons for 5 cents!" When I asked other Apple users their opinion on the Mac Inker and re-inking in general, the feedback was lukewarm at best - comments such as "too messy," "doesn't work," were typical. Since the MacInker cost $42.00 plus $3.00 for a bottle of ink, I didn't want to take a chance, and continued with the above techniques.

In mid October, my folks wanted to know what I wanted for Christmas - so why not, I asked for a MacInker.

Christmas day arrived, and a box about 3/4 the size of a shoe box was under the tree. It contained the MacInker, a squeeze bottle of ink, two re-inking spindles, and three sheets of instructions.

The system is fairly simple. Mac Inker is a plastic platform with several holes the size of quarters in it. There is an electric motor mounted under the platform, with a spindle extending from the motor through the platform. The ribbon snaps onto the platform and spindle, which pulls the ribbon. The heart of the system is the ink spool, which is a hollow cylinder with two O-rings. In between the O-rings are two pin holes. This spool is mounted on the far end of the platform with a wing nut. To my surprise, the spool was fixed, and did not turn.

The set up is simple. First, draw about six inches of ribbon from the cartridge; mount the cartridge on the spindle, and pull the ribbon over the re-inking spool so it fits between the two O-rings. Fill the spool to, 3/4 full of ink from the squeeze ink bottle, and turn on the switch. The ribbon is drawn slowly over the pin holes in the spool, and two parallel lines of ink 1/8" wide are placed on the ribbon as it rolls past at a speed of about three feet per minute.

The instructions emphasize that most new users tend to over ink. To avoid this, they recommend filling the ink reservoir 1/2 to 3/4 full one time, and letting the system run for 3-4 hours

---

uses HyperCard to store a pointer to the window. When the window is created, my XCMD will call back into HyperCard to put a pointer to the window into the global container. Each time the XCMD is called after that, it asks HyperCard for the contents of the container and uses it to access the window. When the window is to be destroyed, my XCMD disposes of the window and sets the container to NIL.

How can one XCMD do all that? Simple. I use an approach similar to that of Hypercard's designers, and modularize everything. Here are the commands I have defined so far:

graphPack 1, PICT number
—open a new window

graphPack 2, x1, y1, x2, y2
—draw a line between the coordinate pairs (local coordinates)

graphPack 3, PICT number
—reinitialize the window and redraw the PICT

graphPack 9
—destroy the window

Using this modular approach, it should be easy for you to add routines to draw boxes, cirlces, etc. If you use my core routines please give me credit in your code. Note that in this project I have not put in extensive error-trapping code: my point was instead to illustrate the use of Callbacks. The only error-checking I do is to beep if an improper number of arguments is passed. It is possible, for example, to crash your system by using a graphPack 9 call without having opened a window using **graphPack 1,1.** Another common source of system crashes is out-of-memory conditions: if your system crashes upon entry to my XCMD, I suggest removing as many INITs and other memory-grabbing things from your system folder. If your routines are crashing, examine them very carefully for minor errors: even a missing asterisk (dereference) can cause a system crash. This leads us directly into a brief discussion on...

Debugging

Debugging XCMDs can be very difficult. It is not possible to use THINK's debugger, since the XCMD will operate only in tandem with Hypercard. It is possible, however, to examine the contents of handles using MacsBug, if you have it installed in your system folder. I use the following code:

```
asm {MOVE.L the_handle, D3}
Debugger();
```

Suppose you want to examine a string in memory while your XCMD is executing. If, for example, D3 contains 26C72, type **dm 26C72.** The address of data object will be stored there (ignore the highest bytes.) **dm (the address)** and you will see the string displayed in ASCII with either a length byte before it (for a Pascal-type string) or a zero byte terminating it (for a C-type string). If the string is not there, your handle is wrong!

If all is well, you can continue execution of the XCMD by typing **g** (for go). If your object is improperly referenced, you will probably have to type **rb** (reboot) to avoid a system crash.

Next time: improving HyperCard's math performance with XFCNs, and using graphPack to graph functions.