Understanding Lightspeed C: Talking Back to HyperCard

Paul R. Potts

April 1989 and March 2025

This is the text of the article I sent to the Washington Apple Pi Journal. I recovered the text in March 2025 from an ancient Microsoft Word for Macintosh file, thanks to Apple's humble TextEdit application, which can open these files and show the text content, when other applications such as Pages cannot, and also to LibreOffice, which can open them up with formatting miraculously intact.

When Washington Apple Pi Journal published this article, they left out all the source code, making it difficult to understand what the article was referring to. I have included the source code below. I had hoped to continue sending the journal additional articles in this series, but after this, I did not send them any more.

In the process of reading over the source code, I found what appears to be a vintage bug in the function **getWindowPtr** *— oops! I have fixed this bug; see the comments in the* ***getWindowPtr*** *function below. Maybe it was ultimately for the best that they did not publish my buggy code.*

In college, I used a more fully-featured version of this code design to build a HyperCard stack that drew multi-colored graphs of functions, to teach basic calculus limits and continuity, in a project for College of Wooster mathematics professor Donald Beane.

Later, working at the Office of Instructional Technology at the University of Michigan, I was able to take the same code design I used for adding additional drawing capabilities to HyperCard, and use it to add additional sound-generating capabilities. I used this to build a HyperCard stack that simulated an audiometer.

In 2025, after reviewing my old code and getting it building and running on the Basilisk II emulator, I decided to take a crack at improving the code and creating a new HyperCard demonstration stack to run it. You can find the results in my article Talking Back to HyperCard Revisited, which contains not just source code, but also links to download my THINK C and HyperCard files.

Talking Back to HyperCard

In my last article 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 discuss how to call back into HyperCard in order to use some of the powerful utility routines available there.

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 that Pascal (and the Toolbox) often uses:

```
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 that this can be done with the following line of code:

```
((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. The line of code is:

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

It 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 to the arguments send back and forth to HyperCard. Fortunately, you don't need to know 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. Now, I want to create a window and draw things in it that are 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.c** and **XCMD.shell.c** in a project with MacTraps. (Do not add the other two source code files to the project: they are brought 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 which will 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 that the user doesn't drag the foreign windows around.

Although HyperCard doesn't know about my window, my XCMD 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 hold 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, circles, 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.

Source code file 1 of 5: working.xcmd.h

```
/*
    File: working.xcmd.h
    Contains a stripped-version of Apple's header information,
    necessary for the graphPack project. Information copyright
    Apple Computer.
*/
typedef struct XCmdBlock {
    short   paramCount;
    Handle  params[16];
    Handle  returnValue;
    Boolean passFlag;

    void    *entryPoint; /* to call back to HyperCard */
    short   request;
    short   result;
    long    inArgs[8];
    long    outArgs[4];
} XCmdBlock, *XCmdBlockPtr;

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

  /*result codes */
```

```
#define xresSucc       0
#define xresFail       1
#define xresNotImp     2

  /* request codes */
#define xreqPasToZero 7
#define xreqZeroToPas 8
#define xreqStrToLong 9
#define xreqStrToNum  10
#define xreqLongToStr 13
#define xreqNumToStr  14
#define xreqGetGlobal 18
#define xreqSetGlobal 19

/* Prototypes of glue routines that we will use. The main program
    must include working.glue.c after its routines. */

extern pascal Handle PasToZero(XCmdBlockPtr paramPtr,
    StringPtr pasStr);
extern pascal void ZeroToPas(XCmdBlockPtr paramPtr
    char *zeroStr, StringPtr pasStr);
extern pascal long StrToLong(XCmdBlockPtr paramPtr,
    Str31 *strPtr);
extern pascal long StrToNum(XCmdBlockPtr paramPtr,
    Str31 *str);
extern pascal void LongToStr(XCmdBlockPtr paramPtr
    long posNum,Str31 *mystr);
extern pascal void NumToStr(XCmdBlockPtr paramPtr
    long num,Str31 *mystr);
extern pascal Handle GetGlobal(XCmdBlockPtr paramPtr,
    StringPtr globName);
extern pascal void SetGlobal(XCmdBlockPtr paramPtr,
    StringPtr globName,Handle globValue);
```

Source code file 2 of 5: working.glue.c

```
/****************************************************************/
/*
    File: working.glue.c
    Contains glue routines for the callbacks used by the
    graphPack project. (modified by Paul Potts)
    Information Copyright Apple Computer
*/

pascal Handle PasToZero(paramPtr,pasStr)
    XCmdBlockPtr paramPtr;
    StringPtr    pasStr;
/* Returns a handle to a zero-terminated string. The caller must
   dispose of the handle. */
{
    paramPtr->inArgs[0] = (long)pasStr;
    paramPtr->request = xreqPasToZero;
    CallPascal(paramPtr->entryPoint);
    return (Handle)paramPtr->outArgs[0];
}

pascal void ZeroToPas(paramPtr,zeroStr,pasStr)
    XCmdBlockPtr paramPtr;
    char         *zeroStr;
    StringPtr    pasStr;
/* Fill the Pascal string with the contents of the zero-terminated
   string.  You create the Pascal string and pass it in as a VAR
   parameter.  Useful for converting the arguments of any XCMD to
   Pascal strings. */
{
    paramPtr->inArgs[0] = (long)zeroStr;
    paramPtr->inArgs[1] = (long)pasStr;
    paramPtr->request = xreqZeroToPas;
    CallPascal(paramPtr->entryPoint);
}

pascal long StrToLong(paramPtr,strPtr)
    XCmdBlockPtr paramPtr;
    Str31 * strPtr;
/* Convert a string of ASCII decimal digits to an unsigned
    long integer. */
{
    paramPtr->inArgs[0] = (long)strPtr;
    paramPtr->request = xreqStrToLong;
    CallPascal(paramPtr->entryPoint);
    return (long)paramPtr->outArgs[0];
}

pascal long StrToNum(paramPtr,str)
    XCmdBlockPtr paramPtr;
    Str31 *      str;
/* Convert a string of ASCII decimal digits to a signed
    long integer. Negative sign is allowed. */
{
    paramPtr->inArgs[0] = (long)str;
    paramPtr->request = xreqStrToNum;
    CallPascal(paramPtr->entryPoint);
    return paramPtr->outArgs[0];
}

pascal void LongToStr(paramPtr,posNum,mystr)
    XCmdBlockPtr paramPtr;
    long         posNum;
    Str31        *mystr;
 /* Convert an unsigned long integer to a Pascal string. Instead of
    returning a new string, as Pascal does, it expects you to create
    mystr and pass it in to be filled. */
{
    paramPtr->inArgs[0] = (long)posNum;
    paramPtr->inArgs[1] = (long)mystr;
    paramPtr->request = xreqLongToStr;
    CallPascal(paramPtr->entryPoint);
}

pascal void NumToStr(paramPtr,num,mystr)
    XCmdBlockPtr paramPtr;
    long  num;
    Str31 *mystr;
 /* Convert a signed long integer to a Pascal string. Instead of
    returning a new string, as Pascal does, it expects you to create
    mystr and pass it in to be filled. */
{
    paramPtr->inArgs[0] = num;
    paramPtr->inArgs[1] = (long)mystr;
    paramPtr->request = xreqNumToStr;
    CallPascal(paramPtr->entryPoint);
}

pascal Handle GetGlobal(paramPtr,globName)
    XCmdBlockPtr paramPtr;
    StringPtr    globName;
/* Return a handle to a zero-terminated string containing the value of
   the specified HyperTalk global variable. */
{
    paramPtr->inArgs[0] = (long)globName;
    paramPtr->request = xreqGetGlobal;
    CallPascal(paramPtr->entryPoint);
    return (Handle)paramPtr->outArgs[0];
}

pascal void SetGlobal(paramPtr,globName,globValue)
    XCmdBlockPtr paramPtr;
    StringPtr globName;
    Handle    globValue;
/* Set the value of the specified HyperTalk global variable to be
   the zero-terminated string in globValue. The contents of the
   Handle are copied, so you must still dispose it afterwards. */
{
    paramPtr->inArgs[0] = (long)globName;
    paramPtr->inArgs[1] = (long)globValue;
    paramPtr->request = xreqSetGlobal;
    CallPascal(paramPtr->entryPoint);
}
```

Source code file 3 of 5: graphPack.c

```
/*****************************************************************/
/* File: graphPack.c                                             */
/* Needs MacHeaders turned on.                                   */
/* This file should be compiled in a project along with the following:
    XCMDshell.c
    the MacTraps library
*/
#include "working.xcmd.h"
#define myStringID 100
/* myStringID is the ID number of the STR resource containing the
    name of the global container in your stack which will hold a
    pointer to the window owned by the XCMD. (Believe me, it's not
    as complicated as it sounds!) */

/* Here are the prototypes for the routines */
GrafPtr getWindowPtr(XCmdBlockPtr paramPtr,
    long strID);
void putWindowPtr(XCmdBlockPtr paramPtr,
    GrafPtr theWindow, long strID);
void trashWindow(XCmdBlockPtr paramPtr);
void makeWindow(XCmdBlockPtr paramPtr);
void makeLine(XCmdBlockPtr paramPtr);
void cleanWindow(XCmdBlockPtr paramPtr);
void SelectCommand(XCmdBlockPtr paramPtr);

/*****************************************************************/
/* This is the dispatcher which determines which one of the
    routines in the graphPack XCMD to run. */

void SelectCommand(paramPtr)
    XCmdBlockPtr paramPtr;
{
    short       which_command;
    Str31Handle command;
    GrafPtr     myWindow, oldWindow;
    GetPort(&oldWindow); /* save Hypercard's port */
    FlushEvents(everyEvent, 0);
    PurgeMem((Size)5000);
    command = (Str31Handle)NewHandle(sizeof(Str31));
    ZeroToPas(paramPtr,(char*)*paramPtr->params[0],
        (StringPtr)*command);
    which_command = (short)StrToNum(paramPtr,*command);
    DisposHandle(command);
    switch (which_command) {
        case 1: if (paramPtr->paramCount == 2)
                    makeWindow (paramPtr);
                else
                    SysBeep(40);
                break;

        case 2: if (paramPtr->paramCount == 5)
                    makeLine (paramPtr);
                else
                    SysBeep(40);
                break;

        case 3: if (paramPtr->paramCount == 2)
                    cleanWindow (paramPtr);
                else
                    SysBeep(40);
                break;

        case 9: if (paramPtr->paramCount == 1)
                    trashWindow (paramPtr);
                else
                    SysBeep(40);
                break;
    }
    FlushEvents (everyEvent, 0);
    SetPort(&oldWindow);
}

/*******************************************************************/

/* This routine will It call back into Hypercard to store the
   location of the window in a container named by the STR
   resource whose ID is passed in.*/

void putWindowPtr(paramPtr, theWindow, strID)
```

```
    XCmdBlockPtr paramPtr;
    GrafPtr theWindow;
    long    strID;
{
    Str31Handle container_name, container_value;
    container_name = (Str31Handle)NewHandle(sizeof(Str31));
    container_value = (Str31Handle)NewHandle(sizeof(Str31));
    container_name = (Str31Handle)GetString(strID);
    LongToStr(paramPtr, (long)theWindow, *container_value);
    container_value = (Str31Handle)PasToZero(paramPtr,
        (StringPtr)*container_value);
    SetGlobal(paramPtr,(StringPtr)*container_name,
        (Handle)container_value);
    ReleaseResource(container_name);
    DisposHandle(container_value);
}

/*****************************************************************/

/* This function will return a pointer to the window owned by
   the XCMD graphPack. It accepts the ID number of the STR
   resource which names the container holding the pointer */

GrafPtr getWindowPtr(paramPtr, strID)
    XCmdBlockPtr paramPtr;
    long         strID;
{
    GrafPtr our_window;
    Str31Handle container_name, container_value, p_str;
    p_str = (Str31Handle)NewHandle(sizeof(Str31));
    container_name = (Str31Handle)GetString(strID);
    container_value = (Str31Handle)GetGlobal(paramPtr,
        (StringPtr)*container_name);
    /* Comment added in 2025: the rest of the function, in the
       version of the code I submitted originally, looked like
       this:

        ZeroToPas(paramPtr,(char*)*container_value,
            (StringPtr)*p_str);
        our_window = (GrafPtr)StrToLong(paramPtr,*p_str);
        return our_window;
        ReleaseResource(container_name);
        DisposHandle(p_str);
        DisposHandle(container_value);

        It looks like I may have put the premature return in
        to work around a crash caused by disposing handles in
        the wrong order, and forgotten to actually debug and
        fix the problem, leaving in place code that leaks. I
        was able to find a slightly later version of the same
        function in my archive of old projects, which ends
        with the lines below. I assume these lines are correct,
        although in 2025 I cannot build and test this code, at
        least not easily.
    */
    ReleaseResource(container_name);
    ZeroToPas(paramPtr,(char*)*container_value,
        (StringPtr)*p_str);
    DisposHandle(container_value);
    our_window = (GrafPtr)StrToLong(paramPtr,*p_str);
    DisposHandle(p_str);
    return our_window;
}

/******************************************************************/

/* This routine will destroy the window owned by the XCMDs and
   free up the memory, then it will set the container to NIL */

void trashWindow(paramPtr)
    XCmdBlockPtr paramPtr;
{
    GrafPtr xcmdWindow;
    xcmdWindow = getWindowPtr(paramPtr, myStringID);
    DisposeWindow(xcmdWindow);
    putWindowPtr(paramPtr, (GrafPtr)0, myStringID);
 }

/******************************************************************/

/* This routine will redraw the window owned by the XCMD. */

void cleanWindow(paramPtr)
    XCmdBlockPtr paramPtr;
{
    GrafPtr     xcmdWindow;
    Str31Handle pict_num;
    PicHandle   myWindowBox;
    long        which_pict;
    xcmdWindow = getWindowPtr(paramPtr, myStringID);
    SetPort(xcmdWindow);
    pict_num = (Str31Handle)NewHandle(sizeof(Str31));
    ZeroToPas(paramPtr,(char*)*paramPtr->params[1],
        (StringPtr)*pict_num);
    which_pict = StrToNum(paramPtr,*pict_num);
    DisposHandle(pict_num);
    myWindowBox = GetPicture(which_pict);
    DrawPicture(myWindowBox, &xcmdWindow->portRect);
    ReleaseResource(myWindowBox);
}

/*******************************************************************/

/* This function  reads two coordinate pairs in order to
   draw a line segment in a window owned by the XCMD. */

void makeLine(paramPtr)
    XCmdBlockPtr paramPtr;
{
    short       horiz, vert, newhoriz, newvert;
    Str31Handle str;
    GrafPtr     myWindow;
    str = (Str31Handle)NewHandle(sizeof(Str31));
    myWindow = getWindowPtr(paramPtr, 100);
    SetPort(myWindow);
    ZeroToPas(paramPtr,(char*)*paramPtr->params[1],
        (StringPtr)*str);
    horiz = (short)StrToNum(paramPtr,*str);
    ZeroToPas(paramPtr,(char*)*paramPtr->params[2],
        (StringPtr)*str);
    vert = (short)StrToNum(paramPtr,*str);
    ZeroToPas(paramPtr,(char*)*paramPtr->params[3],
        (StringPtr)*str);
    newhoriz = (short)StrToNum(paramPtr,*str);
    ZeroToPas(paramPtr,(char*)*paramPtr->params[4],
        (StringPtr)*str);
    newvert = (short)StrToNum(paramPtr,*str);
    DisposHandle(str);
    ForeColor (magentaColor); /* in color on a Mac II */
    PenSize(2,2);
    MoveTo (horiz, vert);
    LineTo (newhoriz, newvert);
}
```

Source code file 4 of 5: makeWindow.c

```
/*******************************************************************/
/* makeWindow.c                                                    */
/* by Paul Potts 11/22/88 */
/* This routine will initialize a dBoxProc window at the location of
   the PICT read, and draw the PICT in it.  To start with a blank
   window, use an empty PICT */

void makeWindow (paramPtr)
    XCmdBlockPtr paramPtr;
{
    long        which_pict;
    Rect        bounding_rect;
    WindowPtr   theWindow;
    PicHandle   myWindowBox;
    Str31Handle pict_num;
    pict_num = (Str31Handle)NewHandle(sizeof(Str31));
    ZeroToPas(paramPtr,(char*)*paramPtr->params[1],
        (StringPtr)*pict_num);
    which_pict = StrToNum(paramPtr,*pict_num);
    myWindowBox = GetPicture(which_pict);
    bounding_rect = (*myWindowBox)->picFrame;
    theWindow = NewWindow (0L, &bounding_rect, "\P", TRUE,
        dBoxProc, -1L, FALSE, 0L);
    SetPort(theWindow);
    DrawPicture(myWindowBox, &theWindow->portRect);
    ReleaseResource(myWindowBox);
    putWindowPtr(paramPtr, theWindow, myStringID);
}
```

Source code file 5 of 5: XCMD.shell.c

```
/********************************************************************/
/* File: XCMD.shell.c                                               */
/* Needs MacHeaders turned on.                                      */
/* This file should be compiled in a project along with the following
files:
    graphPack.c
    the MacTraps library
*/
#include "working.xcmd.h" /* needed to define XCmdBlock, etc */
void SelectCommand(XCmdBlockPtr paramPtr);
pascal void main(XCmdBlockPtr paramPtr);

pascal void main(paramPtr)
    XCmdBlockPtr paramPtr;
{
    SelectCommand(paramPtr);
}
```

```c
#include "working.glue.c"
/********************************************************************/
```