Appendixes

Appendix A

External Commands and Functions

This appendix describes HyperCard's external command and function interface. In addition to general information about external commands and functions, this appendix contains specific information that requires a reading knowledge of Pascal or C to be understood. This appendix does not include information about how to write code, nor does it explain how to use a compiler or assembler to create an executable resource.

Definitions, uses, and examples

External commands and functions are extensions to the HyperTalk built-in command and function set. HyperCard includes interface procedures that make extending HyperTalk in this way convenient and practical for expert programmers.

XCMD and XFCN resources

External commands and functions are executable Macintosh code resources, written in a Macintosh programming language (such as Pascal, C, or 68000 assembly language), which are attached to the HyperCard application or a stack with a resource editor such as ResEdit. The resource type of an external command is 'XCMD' and the resource type of an external function is 'XFCN'. They are often named by their resource types: external commands are termed "ex-commands" (written XCMDs), and external functions are "ex-functions" (written XFCNs).

XCMDs and XFCNs are handled in much the same way by HyperCard: they are separately compiled and attached by a resource mover to stacks or the HyperCard application; they use the object hierarchy in the same way; and they communicate with HyperCard through the same parameter block data structure.

A Macintosh code resource is a compiled (or assembled) executable code module. An 'XCMD' or 'XFCN' resource has no header bytes; it is invoked by a jump instruction to its entry point. These resources are simpler than Macintosh drivers: they can't have any global (or static) data, and they can't be larger than 32K bytes in size. (For more details about these restrictions, see "Guidelines for Writing XCMDs and XFCNs," later in this appendix.)

After they have been created and attached to HyperCard or a stack, external commands and functions are called from HyperTalk in much the same way that built-in commands or user-defined message and function handlers are called.

For detailed information on Macintosh resources, see *Inside Macintosh,* published by Addison-Wesley.

Uses for XCMDs and XFCNs

External commands and functions can provide access to the Macintosh Toolbox and to some of HyperCard's own internal routines; they can provide fast processing speed for time-critical operations; and they can override built-in HyperTalk commands to provide custom solutions. XCMDs or XFCNs can be used for serial port input and output routines, custom search-and-replace routines, color graphics display routines, file input and output routines, and so on.

A typical use for an XCMD would be as an interface for a driver, allowing HyperCard to control an external device such as a videodisc player. Such an interface would have three parts: the driver, the XCMD, and a HyperTalk handler. The driver would be completely separate from HyperCard. (See Volume II of *Inside Macintosh* for information about writing drivers.) The XCMD would be small; its purpose would be to convert HyperTalk messages to the appropriate driver calls. The HyperTalk handler would call the XCMD with various parameters directing it to open or close the driver or to perform a specific control call.

Guidelines for writing XCMDs and XFCNs

XCMDs and XFCNs can call most of the Macintosh Toolbox traps and routines, but they have certain limitations and restrictions. They can't do everything that an application can do because they are guests in HyperCard's heap. In that regard they are more like desk accessories than applications. Here are some guidelines for writing XCMDs and XFCNs:

- Do not initialize the various Macintosh managers by calling their initialization routines. That is, don't call `InitGraf`, `InitFonts`, `InitWindows`, and so on.
- Do not rely upon having lots of RAM available for your XCMD. There is some extra space in HyperCard's heap, but if HyperCard is running in 750K under MultiFinder™, for example, an XCMD should not be bigger than about 32K.
- Do not use register A5 of the 68000-family processor. The value in A5 belongs to HyperCard, and it points to HyperCard's global data, jump table, and other things that constitute an "A5 world." XCMDs do not currently have their own A5 world.
- XCMDs cannot have global data.
- Because they cannot have global data, XCMDs cannot use string literals with MPW C (MPW C makes string literals into global data). To circumvent this restriction, use 'STR ' resources or put the strings in a short assembly-language glue file.
- XCMDs cannot have a jump table, so they cannot have code segments. This restriction imposes a 32K limit on the size of XCMDs for 68000-based machines (the 68020 supports longer branches).
- XCMDs can, however, allocate small chunks of memory by standard `NewHandle` calls. (You can also allocate memory with `NewPtr` calls, but they should be used sparingly to avoid heap fragmentation.)
- If your XCMD allocates some memory in the heap, it should also deallocate the memory.
- If an XCMD allocates a handle to save state information between invocations of the XCMD, then you must pass the handle back to HyperCard to be stored somewhere in the current stack, such as in a hidden field. You must to convert the handle from a long integer to a string, because all values are treated as strings by HyperTalk.
- Since HyperCard jumps blindly to the start of an XCMD's code, it is important that the main routine actually ends up at the start of the XCMD. In other words, the XCMD glue must follow the main routine, so the link order is vitally important.
- If, as you write, the size of your XCMD begins to approach 32K, consider converting it to a driver.

Flash: an example XCMD

An example external command included with HyperCard is `flash`, which inverts the screen display (changes the black pixels to white and vice versa) a specified number of times. A version of `flash` written and compiled in MPW Pascal has already been attached to the HyperCard application file (that is, to HyperCard itself).

`Flash` is invoked from HyperCard just like a HyperTalk command. That is, you send the message `flash` to HyperCard from the Message box or from an executing script. The flash message takes one parameter: an integer. The `flash` XCMD inverts the screen display twice that many times. For example, the following handler, in response to a `mouseUp` message, sends the `flash` message and its parameter. When the message reaches HyperCard, it invokes the `flash` external command, which inverts the screen display 20 times:

```
on mouseUp
  flash 10
end mouseUp
```

The screen display flashes (is inverted and inverted back again) 10 times.

Flash listing in MPW Pascal

Here's the Pascal listing for `flash`:

```
(*
 * Flash.p - A sample HyperCard XCMD to highlight the screen
 *         - Copyright Apple Computer, Inc.  1987-1988.
 *         - All Rights Reserved.
 *
 * Build instructions:
 *
 * Pascal Flash.p -o Flash.p.o
 * Link Flash.p.o -sg Flash -rt XCMD=0 -m ENTRYPOINT -o StackName
 *
 *)

{$R-}


{$S Flash } { Segment name must be same as command name }
```

```
(*
 * DummyUnit is what HyperTalk jumps to when running the XCMD.
 * Also note that XCMDs do not currently support their own A5 World,
 * thus NO GLOBAL VARIABLES are allowed.  If the link fails then that
 * means the Pascal compiler generated A5-relative code.  (This may
 * happen if you try to use the Pascal libraries, for example.)
 *
 *)

UNIT DummyUnit;

INTERFACE

USES MemTypes, QuickDraw, HyperXCmd;

PROCEDURE EntryPoint(paramPtr: XCmdPtr);

IMPLEMENTATION

TYPE Str31 = String[31];

PROCEDURE Flash(paramPtr: XCmdPtr); FORWARD;

   PROCEDURE EntryPoint(paramPtr: XCmdPtr);
   BEGIN
     Flash(paramPtr);
   END;

   PROCEDURE Flash(paramPtr: XCmdPtr);
   VAR flashCount:  INTEGER;
     again:         INTEGER;
     port:          GrafPtr;
     str:           Str255;
     when:          LongInt;
     ticksPtr:      ^LongInt;

   {$I XCmdGlue.inc }
```

```
  BEGIN
    ZeroToPas(paramPtr^.params[1]^,str); { first param is flash count }
    flashCount := StrToNum(str);
    GetPort(port);
    ticksPtr := Pointer($16A);

    IF (paramPtr^.paramCount <> 1) OR (flashCount < 1)
    THEN flashCount := 3;

    FOR again := 1 TO 2 * flashCount DO
      BEGIN
        when := ticksPtr^ + 4;
        InvertRect(port^.portRect);
        REPEAT UNTIL ticksPtr^ >= when;
      END;
  END;

END.
```

Flash listing in MPW C

Here's a version of `flash` written in MPW C:

```
/*
 *   Flash.c - A sample HyperCard XCMD to highlight the screen
 *             -    Copyright Apple Computer, Inc.  1987-1988.
 *             -    All Rights Reserved.
 *
 *   Build instructions:
 *
 *   C  Flash.c -o Flash.c.o
 *   Link Flash.c.o -sg CFlash -rt XCMD=5 -o StackName
 *
 */

#define __SEG__ CFlash /* Segment name must be the same as command name */

#include <HyperXCmd.h> /* HT interface and #includes Types.h, Memory.h */
#include <QuickDraw.h>

pascal void MacsBug() extern 0xA9FF; /* useful for debugging */
```

```
/*
 *   Your routine MUST be the first code that is generated in the file, as
 *   HyperTalk simply JSRs to the start of the XCMD segment in memory.
 *  Therefore the XCmdGlue.c file must be included after the main routine,
 *   being CFlash in this sample XCMD.  Also note that XCMDs do not currently
 *   support their own A5 World, thus NO GLOBAL VARIABLES are allowed.
 *  If the link fails then that means the C compiler generated A5-relative
 *   code.  (This happens if you try to use the C libraries or use strings
 *   in the code.  Use a STR resource instead.)
 *
 */

pascal void CFlash(paramPtr)
     XCmdBlockPtr paramPtr;
{
     short flashCount,again;
     GrafPtr port;
     Str255 str;

     ZeroToPas(paramPtr,*(paramPtr->params[0]),&str); /* get flash count */
     flashCount = StrToNum(paramPtr,&str);   /* convert to num */
     if (paramPtr->paramCount != 1) flashCount = 3;  /* default if no param */
     if (flashCount < 1) flashCount = 3;    /* must be positive */
     GetPort(&port);
     for (again = 1; again <= flashCount; again++) {
          InvertRect(&port->portRect);
          InvertRect(&port->portRect);
     }
}

#include <XCmdGlue.c> /* C routines for HyperCard callbacks */
```

Flash listing in 68000 assembly language

Here's the 68000 assembly language listing for `flash`:

```
*
* Flash.a    - A sample HyperCard XCMD in 68000 Assembly
*            - Copyright Apple Computer, Inc.  1988.
*            - All Rights Reserved.
*
* Build Instructions:
*
*           Asm  Flash.a -o Flash.a.o
*           Link Flash.a.o -sg AFlash -rt XCMD=7 -o StackName
*
*
            INCLUDE 'QuickEqu.a'
            INCLUDE 'Traps.a'

            SEG 'AFlash'                    ; Segname must be same as command name

AFlash      PROC                            ; uses a0,a1,d1
            link    a6,#-4
            move.l  d4,-(sp)                ; save
            move.l  8(a6),a0                ; get paramPtr in a temp reg
            move.l  2(a0),a1                ; get handle to flashCount (as c string)
            move.l  (a1),a1                 ; deref
            move.w  #3,d4                   ; StrToNum default result

@1          move.b  (a1)+,d1                ; get a char
            cmp.b   #'0',d1                 ; test for a number
            blt.s   @2                      ; less than valid
            cmp.b   #'9',d1
            bgt.s   @2                      ; greater than valid

            and.w   #$000F,d1               ; mask to value of legal char
            move.w  d1,d4                   ; stick value into result

@2          pea     -4(a6)                  ; var result of GetPort
            _GetPort
            bra.s   @4                      ; get into DBRA loop

@3          move.l  -4(a6),a0               ; get port
            pea     portRect(a0)            ; address of portRect
            _InverRect
            move.l  -4(a6),a0               ; get port
            pea     portRect(a0)            ; address of portRect
            _InverRect
```

```
@4          dbra    d4,@3

            move.l  (sp)+,d4                    ; restore
            unlk    a6

            move.l  (sp)+,a0                    ; rts Pascal style
            add.l   #4,a7
            jmp     (a0)

            END
```

Peek: an example XFCN

An example external function is `peek`, which returns the contents of a memory location whose address is passed with the function call. `Peek` is not already attached to the HyperCard application like the `flash` XCMD; you must compile it yourself and attach it to HyperCard or a stack with a resource editor like ResEdit (see "Attaching an XCMD or XFCN" later in this appendix).

Peek listing in MPW Pascal

Here's the Pascal listing for `peek`:

```
(*
 * Peek.p - A sample HyperCard XFCN to return the contents of memory
 *          - Copyright Apple Computer, Inc. 1987,1988.
 *          - All Rights Reserved.
 *
 * Build instructions:
 *
 * Pascal Peek.p -o Peek.p.o
 * Link Peek.p.o -sg Peek -rt XFCN=1 -m ENTRYPOINT -o StackName
 *
 *)

{$R-}

{$S Peek } { Segment name must be same as command name }

(*
 * DummyUnit is what HyperTalk jumps to when running the XCFN.
 * Also note that XCFNs do not currently support their own A5 World,
 * thus NO GLOBAL VARIABLES are allowed.  If the link fails then that
 * means the Pascal compiler generated A5-relative code.  (This may
 * happen if you try to use the Pascal libraries, for example.)
 *
 *)
```

```
UNIT DummyUnit;

INTERFACE

USES MemTypes, HyperXCmd;

PROCEDURE EntryPoint(paramPtr: XCmdPtr);

IMPLEMENTATION

TYPE Str31 = String[31];
          WordPtr = ^INTEGER;
          LongPtr = ^LongInt;

PROCEDURE Peek(paramPtr: XCmdPtr); FORWARD;

   PROCEDURE EntryPoint(paramPtr: XCmdPtr);
   BEGIN
     Peek(paramPtr);
   END;

   PROCEDURE Peek(paramPtr: XCmdPtr);
   VAR peekAddr,peekSize,peekVal: LongInt;
        str: Str255;

   {$I XCmdGlue.inc }
```

```
  BEGIN
    WITH paramPtr^ DO
      BEGIN
        { first param is addr }
        ZeroToPas(params[1]^,str);
        peekAddr := StrToNum(str);

        { second param, if given, is size }
        peekSize := 1;
        IF paramCount = 2 THEN
          BEGIN
            ZeroToPas(params[2]^,str);
            peekSize := StrToNum(str);
          END;

        CASE peekSize OF
          1: peekVal := BAND($000000FF,Ptr(peekAddr)^);
          2: peekVal := BAND($0000FFFF,WordPtr(BAND($FFFFFFFE,peekAddr))^);
          4: peekVal := LongPtr(BAND($FFFFFFFE,peekAddr))^;
          OTHERWISE peekVal := 0;
        END;

        str := NumToStr(peekVal);
        returnValue := PasToZero(str);
      END;
  END;

END.
```

Peek listing in MPW C

Here's the MPW C code listing for peek:

```
/*
 *   Peek.c -  A sample HyperCard XFCN to return the contents of memory
 *             -    Copyright Apple Computer, Inc.  1987,1988.
 *             -    All Rights Reserved.
 *
 *   Build instructions:
 *
 *   C Peek.c -o Peek.c.o
 *   Link Peek.c.o -sg CPeek -rt XFCN=6 -o StackName
 *
 */

#define __SEG__ CPeek /* Segment name must be the same as command name */

#include <HyperXCmd.h> /* HT interface and #includes Types.h, Memory.h */

pascal void MacsBug() extern 0xA9FF; /* useful for debugging */

#define PEEKBYTE(address) *((char *) address)
#define PEEKWORD(address) *((short *) address)
#define PEEKLONG(address) *((long *) address)

/*
 *   Your routine MUST be the first code that is generated in the file, as
 *   HyperTalk simply JSRs to the start of the XFCN segment in memory.
 *  Therefore the XCmdGlue.c file must be included after the main routine,
 *   being CPeek in this sample XFCN.  Also note that XFCNs do not currently
 *   support their own A5 World, thus NO GLOBAL VARIABLES are allowed.
 *  If the link fails then that means the C compiler generated A5-relative
 *   code.  (This happens if you try to use the C libraries or use strings
 *   in the code.  Use a STR resource instead.)
 *
 */
```

```
pascal void CPeek(paramPtr)
     XCmdBlockPtr paramPtr;
{
     char str[256];
     short argc;
     long peekAddr, peekSize, peekVal;
     Handle argv1, argv2;

     argc  = paramPtr->paramCount;
     argv1 = paramPtr->params[0];
     argv2 = paramPtr->params[1];

     ZeroToPas(paramPtr, *argv1, str); /* CtoP string */
     peekAddr = StrToNum(paramPtr, str); /* get address */

     if (argc == 2) {
          ZeroToPas(paramPtr, *argv2, str); /* CtoP string */
          peekSize = StrToNum(paramPtr, str); /* get size */
     }
     else
          peekSize = 1;

     switch(peekSize) {
          case 1:   peekVal = PEEKBYTE(peekAddr); break;
          case 2:   peekVal = PEEKWORD(peekAddr); break;
          case 4:   peekVal = PEEKLONG(peekAddr); break;
          default:  peekVal = 0;
     }
     NumToStr(paramPtr, peekVal, str);

     /* XFCN: make sure to return a result, the only change from an XCMD */
     paramPtr->returnValue = PasToZero(paramPtr, str);
}

#include <XCmdGlue.c>
```

Peek is invoked just like a user-defined function handler. That is, you put the function name in a HyperTalk statement followed by one argument within parentheses—an integer representing the memory location whose contents you want HyperCard to return. For example:

```
on mouseUp
  put peek(0) into msg
end mouseUp
```

The current contents of memory address 0 are displayed in the Message box.

Accessing an XCMD or XFCN

You access XCMDs and XFCNs from HyperTalk using the regular message syntax and user-defined function call syntax. The message or function call is passed through the HyperCard object hierarchy.

Invoking XCMDs and XFCNs

You invoke an XCMD as you do a message handler. That is, you type the name of the XCMD followed by its parameters in a HyperTalk script or in the Message box. Separate the parameters (if there are more than one) with commas, and put quotation marks around parameters of more than one word. When the script executes or when you send the Message box contents by pressing Return or Enter, HyperCard sends the message through the normal object hierarchy. For external commands, the Macintosh resource name correlates to the message name—the first word in the message.

Similarly, you call an XFCN in a HyperTalk statement in the same way you would a user-defined function (use parentheses rather than the word `the`), which calls a function handler somewhere farther along the hierarchy. Enclose any parameters within parentheses, separate them (if more than one) with commas, and put quotation marks around parameters of more than one word. If the function takes no parameters, append empty parentheses after it. For external functions, the Macintosh resource name correlates to the function name—the word preceding parentheses in the function call.

You can pass a maximum of 16 parameters to an XCMD or XFCN.

Object hierarchy

External commands and functions use the object hierarchy in the same way as message and function handlers and built-in commands and functions. External commands and functions can be attached to any stack or to the HyperCard application.

If a stack receives a message or function call for which it has no handler, then before passing the message or function call to the next object, it checks to see if it has an external command or function of the same name. When HyperCard receives a message or function call, it checks to see if it has an external command or function *before* it looks for a built-in command or function.

That is, HyperCard searches for message and function handlers, XCMDs and XFCNs, and built-in commands and functions through the hierarchy shown in Figure A-1.

Chapter 2 discusses the message-passing hierarchy, including the dynamic path, in detail.

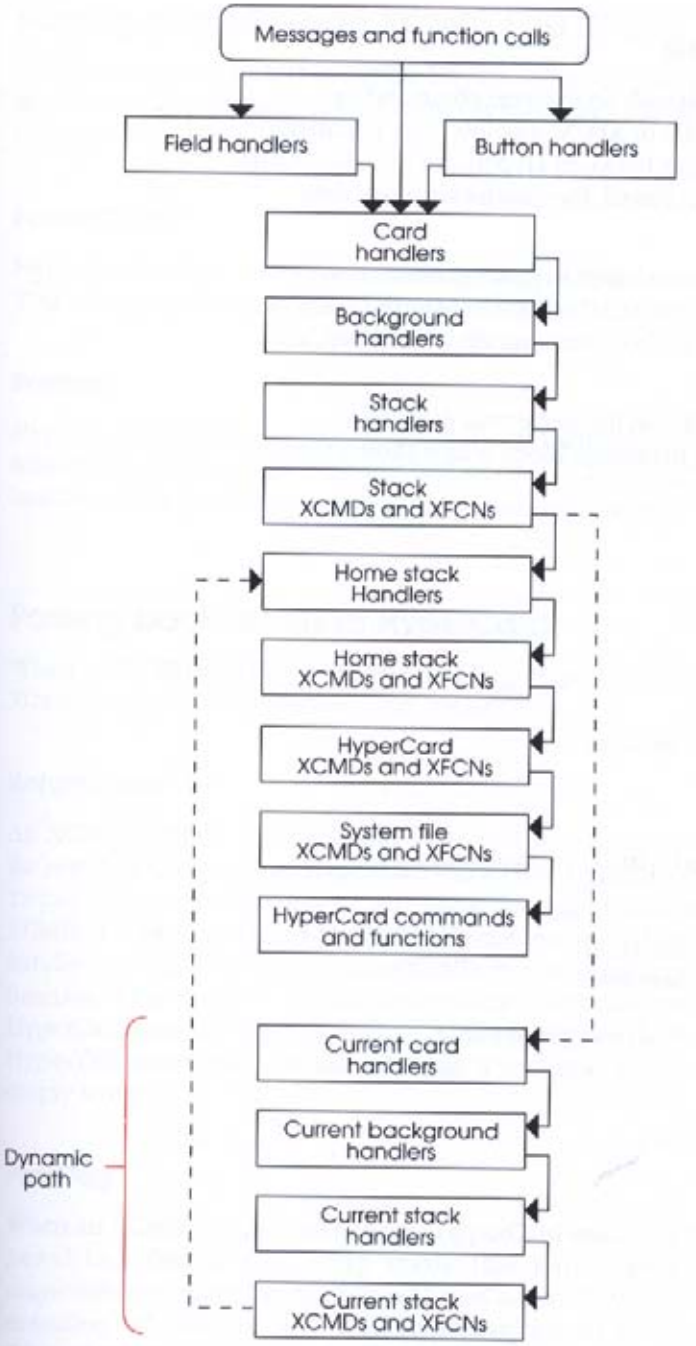

Figure A-1
Message-passing hierarchy, including XCMDs and XFCNs

Parameter block data structure

If HyperCard matches a message or function call with an external command or function, it passes a single argument to the XCMD or XFCN: a pointer to a parameter block called an `XCmdBlock`. All communication between HyperCard and the XCMD or XFCN passes through the parameter block. In Pascal, the parameter block data structure is a `record`; in C it's a `struct`.

HyperCard uses the first two fields of the parameter block to pass information to the XCMD or XFCN before invoking its execution. The XCMD or XFCN uses the other data fields in the `XCmdBlock` to pass back results and to communicate with HyperCard during execution.

The parameter block is listed in both Pascal and C in the respective definition interface files later in this appendix. The Pascal parameter block is also shown here for convenience:

```
TYPE

   XCmdPtr = ^XCmdBlock;
   XCmdBlock =
      RECORD
         paramCount:  INTEGER;
         params:       ARRAY[1..16] OF Handle;
         returnValue: Handle;
         passFlag:   BOOLEAN;

         entryPoint:  ProcPtr;  { to call back to HyperCard }
         request:      INTEGER;
         result:       INTEGER;
         inArgs:       ARRAY[1..8] OF LongInt;
         outArgs:      ARRAY[1..4] OF LongInt;
      END;

END;
```

Passing parameters to XCMDs and XFCNs

Before calling the XCMD or XFCN, HyperCard places the number of parameters and handles to the parameter strings in two fields of the `XCmdBlock:` `paramCount` and `params`.

ParamCount

HyperCard puts an integer representing the parameter count in field `paramCount`. You can pass a maximum of 16 parameter strings.

Params

HyperCard evaluates the parameters and puts their values into memory as zero-terminated ASCII strings. Before it invokes the XCMD or XFCN, HyperCard puts the handles to the parameter strings into the `params` array.

Passing back results to HyperCard

When an XCMD or XFCN finishes executing, HyperCard examines two fields of the `XCmdBlock:` `returnValue` and `passFlag`.

ReturnValue

An XCMD or XFCN can optionally store one zero-terminated string to communicate the result of its execution. HyperCard will look for a handle to the result string in the `returnValue` field of the `XCmdBlock`. Storing a result string is optional for an XCMD; it is expected of an XFCN, but it's not required. If you store a result string handle into `returnValue` in an XCMD, the user can get it by using the HyperTalk function `the result` (useful for explaining why there was an error). For an XFCN, HyperCard uses the `returnValue` string to replace the function call itself in the HyperTalk statement containing the call. If you don't store anything, the result is the empty string.

PassFlag

When an XCMD or XFCN terminates, HyperCard examines the Boolean value of the `passFlag` field. If `passFlag` is false (the normal case), control passes back to the previously executing handler (or to HyperCard's idle state if no handler was executing). If `passFlag` is true, HyperCard passes the message or function call to the next object in the hierarchy. This has the same effect as the `pass` control statement in a script.

Callbacks

The remaining five fields of the `XCmdBlock` record have to do with calling HyperCard back in the middle of execution of an XCMD or XFCN. You use the callback mechanism to obtain data or request HyperCard to perform an action. HyperCard has 29 callback requests (see "Request Codes" later in this appendix). The five `XCmdBlock` fields that compose the callback interface are `entryPoint`, `request`, `result`, `inArgs`, and `outArgs`.

EntryPoint

When HyperCard sets up the `XCmdBlock` data structure before passing control to an XCMD or XFCN, it places an address in `entryPoint`. The XCMD or XFCN uses this address to execute a jump instruction to pass control to HyperCard for the callback.

Request

Before executing the jump instruction, the XCMD or XFCN puts an integer representing the callback request it's making into the `request` field. The request codes are listed in "Callback Procedures and Functions" later in this appendix.

Result

After it completes the callback request, HyperCard places an integer result code in the `result` field. The result code can be 0, 1, or 2. If the callback executed successfully, the result is 0; if it failed, the result is 1; if the callback request is not implemented in HyperCard, the result is 2.

InArgs

The XCMD or XFCN sends up to eight arguments to HyperCard as long integers in the `inArgs` array. Depending on the callback request, HyperCard expects arguments in certain elements of the `inArgs` array. In many callbacks, the arguments are pointers to zero-terminated strings. The callback arguments are shown in Pascal in "Callback Procedures and Functions" later in this appendix.

OutArgs

After it executes the callback request, HyperCard returns up to four long integers (or other types, such as handles) to the XCMD or XFCN as elements of the `outArgs` array. The arguments HyperCard returns from callbacks are shown in Pascal in "Callback Procedures and Functions" later in this appendix.

Callback procedures and functions

If you want to manage a callback to HyperCard yourself, you can define the `XCmdBlock` data structure in your XCMD or XFCN. Then you can put values you want to send to HyperCard in `inArgs`, put a request code in `request`, and execute a jump instruction to the address HyperCard places in `entryPoint`. HyperCard returns values in `outArgs` and a result code in `result`.

However, if you use MPW Pascal or C, you can take advantage of interface definition and "glue" files. (The definition and glue files are listed later in this appendix and are also available on disk from APDA, the Apple Programmer's and Developer's Association. Information about APDA is listed at the end of this appendix.) The definition and glue files provide simple procedure and function calls that you can use inside your XCMD or XFCN to handle callback requests more easily. Include them when you compile your XCMD or XFCN.

The Pascal code for an XCMD or XFCN should include the definition file `HyperXCmd.p` at the beginning of the `USES` clause and the glue file `XCmdGlue.inc` at the end with the `$I` directive. There must be an argument of type `XCmdPtr` passed by HyperCard to the XCMD or XFCN. In the glue routines, all strings are Pascal strings unless noted as zero-terminated strings (which have no length byte; the end of the string is indicated by a null byte). In general, if a handle is returned, the XCMD or XFCN is responsible for disposing of it.

Definition interface files

The MPW Pascal definition interface file is `HyperXCmd.p`. The MPW C definition interface file is `HyperXCmd.h`. These files define the `XCmdBlock` parameter block described earlier in this appendix. They also define the constants representing the callback result codes and request codes.

Definition file in MPW Pascal

The interface definition file in MPW Pascal is as follows:

```
(*
 * HyperXCmd.p          - Interface to HyperTalk callback routines
 *                      - Copyright Apple Computer, Inc. 1987,1988.
 *                      - All Rights Reserved.
 *
 *)

UNIT HyperXCmd;
```

```
INTERFACE

CONST

   { result codes }
   xresSucc       = 0;
   xresFail       = 1;
   xresNotImp     = 2;

   { request codes }
   xreqSendCardMessage   =  1;
   xreqEvalExpr          =  2;
   xreqStringLength      =  3;
   xreqStringMatch       =  4;
   xreqSendHCMessage     =  5;
   xreqZeroBytes         =  6;
   xreqPasToZero         =  7;
   xreqZeroToPas         =  8;
   xreqStrToLong         =  9;
   xreqStrToNum          = 10;
   xreqStrToBool         = 11;
   xreqStrToExt          = 12;
   xreqLongToStr         = 13;
   xreqNumToStr          = 14;
   xreqNumToHex          = 15;
   xreqBoolToStr         = 16;
   xreqExtToStr          = 17;
   xreqGetGlobal         = 18;
   xreqSetGlobal         = 19;
   xreqGetFieldByName    = 20;
   xreqGetFieldByNum     = 21;
   xreqGetFieldByID      = 22;
   xreqSetFieldByName    = 23;
   xreqSetFieldByNum     = 24;
   xreqSetFieldByID      = 25;
   xreqStringEqual       = 26;
   xreqReturnToPas       = 27;
   xreqScanToReturn      = 28;
   xreqScanToZero        = 39;
```

```
TYPE

   XCmdPtr = ^XCmdBlock;
   XCmdBlock =
      RECORD
         paramCount:       INTEGER;
         params:           ARRAY[1..16] OF Handle;
         returnValue:      Handle;
         passFlag:         BOOLEAN;

         entryPoint:   ProcPtr;  { to call back to HyperCard }
         request:          INTEGER;
         result:           INTEGER;
         inArgs:           ARRAY[1..8] OF LongInt;
         outArgs:          ARRAY[1..4] OF LongInt;
      END;

END;
```

Definition file in MPW C

The interface definition file in MPW C includes the parameter block definition, the result and request code constants, and forward definitions for the glue routines. The definition file is as follows:

```
/*
 *   HyperXCmd.h    - Interfaces for HyperTalk callback routines
 *                  - Copyright Apple Computer, Inc. 1987,1988.
 *                  - All Rights Reserved.
 *
 *   #include this file before your program.
 *   #include "XCmdGlue.c" after your code.
 *
 */

#include <Types.h>
#include <Memory.h>

pascal void Debugger() extern 0xA9FF;
```

```
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;


/* result codes */
#define   xresSucc       0
#define   xresFail       1
#define   xresNotImp     2

/* request codes */
#define   xreqSendCardMessage          1
#define   xreqEvalExpr                 2
#define   xreqStringLength             3
#define   xreqStringMatch              4
#define   xreqSendHCMessage            5
#define   xreqZeroBytes                6
#define   xreqPasToZero                7
#define   xreqZeroToPas                8
#define   xreqStrToLong                9
#define   xreqStrToNum                 10
#define   xreqStrToBool                11
#define   xreqStrToExt                 12
#define   xreqLongToStr                13
#define   xreqNumToStr                 14
#define   xreqNumToHex                 15
#define   xreqBoolToStr                16
#define   xreqExtToStr                 17
#define   xreqGetGlobal                18
```

```
#define   xreqSetGlobal              19
#define   xreqGetFieldByName         20
#define   xreqGetFieldByNum          21
#define   xreqGetFieldByID           22
#define   xreqSetFieldByName         23
#define   xreqSetFieldByNum          24
#define   xreqSetFieldByID           25
#define   xreqStringEqual            26
#define   xreqReturnToPas            27
#define   xreqScanToReturn           28
#define   xreqScanToZero             39   /* was supposed to be 29!  Oops! */

/* Forward definitions of glue routines.  Main program
     must include XCmdGlue.c after its routines.  */

pascal void SendCardMessage(paramPtr,msg)
     XCmdBlockPtr        paramPtr;      StringPtr      msg;      extern;
pascal Handle EvalExpr(paramPtr,expr)
     XCmdBlockPtr        paramPtr;      StringPtr      expr;     extern;
pascal long StringLength(paramPtr,strPtr)
     XCmdBlockPtr        paramPtr;      StringPtr      strPtr;   extern;
pascal Ptr StringMatch(paramPtr,pattern,target)
     XCmdBlockPtr        paramPtr;      StringPtr      pattern;
     Ptr  target; extern;
pascal void SendHCMessage(paramPtr,msg)
     XCmdBlockPtr        paramPtr;      StringPtr      msg;      extern;
pascal void ZeroBytes(paramPtr,dstPtr,longCount)
     XCmdBlockPtr        paramPtr;      Ptr  dstPtr;
     long      longCount;     extern;
pascal Handle PasToZero(paramPtr,pasStr)
     XCmdBlockPtr        paramPtr;      StringPtr      pasStr;   extern;
pascal void ZeroToPas(paramPtr,zeroStr,pasStr)
     XCmdBlockPtr        paramPtr;      char           *zeroStr;
     StringPtr pasStr;       extern;
pascal long StrToLong(paramPtr,strPtr)
     XCmdBlockPtr        paramPtr;      Str31 *        strPtr;   extern;
pascal long StrToNum(paramPtr,str)
     XCmdBlockPtr        paramPtr;      Str31 *        str;      extern;
pascal Boolean StrToBool(paramPtr,str)
     XCmdBlockPtr        paramPtr;      Str31 *        str;      extern;
pascal void StrToExt(paramPtr,str,myext)
     XCmdBlockPtr        paramPtr;      Str31 *        str;
     extended *     myext;     extern;
```

```
pascal void LongToStr(paramPtr,posNum,mystr)
     XCmdBlockPtr        paramPtr;       long       posNum;
     Str31 *   mystr;    extern;
pascal void NumToStr(paramPtr,num,mystr)
     XCmdBlockPtr        paramPtr;       long       num;
     Str31 *   mystr;    extern;
pascal void NumToHex(paramPtr,num,nDigits,mystr)
     XCmdBlockPtr        paramPtr;       long       num;
     short     nDigits;                  Str31 * mystr;      extern;
pascal void BoolToStr(paramPtr,bool,mystr)
     XCmdBlockPtr        paramPtr;       Boolean    bool;
     Str31 *   mystr;    extern;
pascal void ExtToStr(paramPtr,myext,mystr)
     XCmdBlockPtr        paramPtr;       extended * myext;
     Str31 *   mystr;    extern;
pascal Handle GetGlobal(paramPtr,globName)
     XCmdBlockPtr        paramPtr;       StringPtr      globName; extern;
pascal void SetGlobal(paramPtr,globName,globValue)
     XCmdBlockPtr        paramPtr;       StringPtr      globName;
     Handle    globValue;     extern;
pascal Handle GetFieldByName(paramPtr,cardFieldFlag,fieldName)
     XCmdBlockPtr        paramPtr;       Boolean    cardFieldFlag;
     StringPtr      fieldName;     extern;
pascal Handle GetFieldByNum(paramPtr,cardFieldFlag,fieldNum)
     XCmdBlockPtr        paramPtr;       Boolean    cardFieldFlag;
     short     fieldNum;      extern;
pascal Handle GetFieldByID(paramPtr,cardFieldFlag,fieldID)
     XCmdBlockPtr        paramPtr;       Boolean    cardFieldFlag;
     short     fieldID;       extern;
pascal void SetFieldByName(paramPtr,cardFieldFlag,fieldName,fieldVal)
     XCmdBlockPtr        paramPtr;       Boolean    cardFieldFlag;
     StringPtr      fieldName;     Handle    fieldVal;      extern;
pascal void SetFieldByNum(paramPtr,cardFieldFlag,fieldNum,fieldVal)
     XCmdBlockPtr        paramPtr;       Boolean    cardFieldFlag;
     short     fieldNum;                 Handle     fieldVal;      extern;
pascal void SetFieldByID(paramPtr,cardFieldFlag,fieldID,fieldVal)
     XCmdBlockPtr        paramPtr;       Boolean    cardFieldFlag;
     short     fieldID;                  Handle     fieldVal;      extern;
```

```
pascal Boolean StringEqual(paramPtr,str1,str2)
     XCmdBlockPtr       paramPtr;      Str31 *       str1;
     Str31 *       str2;     extern;
pascal void ReturnToPas(paramPtr,zeroStr,pasStr)
     XCmdBlockPtr       paramPtr;      Ptr       zeroStr;
     StringPtr     pasStr;   extern;
pascal void ScanToReturn(paramPtr,scanHndl)
     XCmdBlockPtr       paramPtr;      Ptr *     scanHndl;     extern;
pascal void ScanToZero(paramPtr,scanHndl)
     XCmdBlockPtr       paramPtr;      Ptr *     scanHndl;     extern;
```

Glue routines

The MPW Pascal callback glue routines file is `XCmdGlue.inc`. The MPW C definition file is `XCmdGlue.c`. These files define the interface procedures and functions that handle callback requests for XCMDs and XFCNs written in the same language. The first line of each procedure or function definition shows the name and parameters that you use to call it.

Glue routines in MPW Pascal

The first procedure defines the jump instruction with which the XCMD or XFCN passes control to HyperCard to carry out its callback request. The MPW Pascal glue routines are as follows:

```
(*
 * XCMDGlue.inc - Implementation of HyperTalk callback routines
 *                - Copyright Apple Computer, Inc. 1987,1988.
 *                - All Rights Reserved.
 *
 *)

{ Assumes the XCMD has included this file and
  has named its argument "paramPtr" }
```

```
PROCEDURE DoJsr(addr: ProcPtr); INLINE $205F,$4E90;


FUNCTION StringMatch(pattern: Str255; target: Ptr): Ptr;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@pattern);
         inArgs[2] := ORD(target);
         request := xreqStringMatch;
         DoJsr(entryPoint);
         StringMatch := Ptr(outArgs[1]);
      END;
END;


FUNCTION PasToZero(str: Str255): Handle;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@str);
         request := xreqPasToZero;
         DoJsr(entryPoint);
         PasToZero := Handle(outArgs[1]);
      END;
END;


PROCEDURE ZeroToPas(zeroStr: Ptr; VAR pasStr: Str255);
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(zeroStr);
         inArgs[2] := ORD(@pasStr);
         request := xreqZeroToPas;
         DoJsr(entryPoint);
      END;
END;
```

```
FUNCTION StrToLong(str: Str31): LongInt;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@str);
         request := xreqStrToLong;
         DoJsr(entryPoint);
         StrToLong := outArgs[1];
      END;
END;


FUNCTION StrToNum(str: Str31): LongInt;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@str);
         request := xreqStrToNum;
         DoJsr(entryPoint);
         StrToNum := outArgs[1];
      END;
END;


FUNCTION StrToBool(str: Str31): BOOLEAN;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@str);
         request := xreqStrToBool;
         DoJsr(entryPoint);
         StrToBool := BOOLEAN(outArgs[1]);
      END;
END;


FUNCTION StrToExt(str: Str31): Extended;
VAR x: Extended;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@str);
         inArgs[2] := ORD(@x);
         request := xreqStrToExt;
         DoJsr(entryPoint);
         StrToExt := x;
      END;
END;
```

```
FUNCTION LongToStr(posNum: LongInt): Str31;
VAR str: Str31;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := posNum;
         inArgs[2] := ORD(@str);
         request := xreqLongToStr;
         DoJsr(entryPoint);
         LongToStr := str;
      END;
END;


FUNCTION NumToStr(num: LongInt): Str31;
VAR str: Str31;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := num;
         inArgs[2] := ORD(@str);
         request := xreqNumToStr;
         DoJsr(entryPoint);
         NumToStr := str;
      END;
END;


FUNCTION NumToHex(num: LongInt; nDigits: INTEGER): Str31;
VAR str: Str31;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := num;
         inArgs[2] := nDigits;
         inArgs[3] := ORD(@str);
         request := xreqNumToHex;
         DoJsr(entryPoint);
         NumToHex := str;
      END;
END;
```

```
FUNCTION ExtToStr(num: Extended): Str31;
VAR str: Str31;
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(@num);
       inArgs[2] := ORD(@str);
       request := xreqExtToStr;
       DoJsr(entryPoint);
       ExtToStr := str;
     END;
END;


FUNCTION BoolToStr(bool: BOOLEAN): Str31;
VAR str: Str31;
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := LongInt(bool);
       inArgs[2] := ORD(@str);
       request := xreqBoolToStr;
       DoJsr(entryPoint);
       BoolToStr := str;
     END;
END;


PROCEDURE SendCardMessage(msg: Str255);
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(@msg);
       request := xreqSendCardMessage;
       DoJsr(entryPoint);
     END;
END;


PROCEDURE SendHCMessage(msg: Str255);
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(@msg);
       request := xreqSendHCMessage;
       DoJsr(entryPoint);
     END;
END;
```

```
FUNCTION EvalExpr(expr: Str255): Handle;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@expr);
         request := xreqEvalExpr;
         DoJsr(entryPoint);
         EvalExpr := Handle(outArgs[1]);
      END;
END;


FUNCTION StringLength(strPtr: Ptr): LongInt;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(strPtr);
         request := xreqStringLength;
         DoJsr(entryPoint);
         StringLength := outArgs[1];
      END;
END;


FUNCTION GetGlobal(globName: Str255): Handle;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@globName);
         request := xreqGetGlobal;
         DoJsr(entryPoint);
         GetGlobal := Handle(outArgs[1]);
      END;
END;


PROCEDURE SetGlobal(globName: Str255; globValue: Handle);
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@globName);
         inArgs[2] := ORD(globValue);
         request := xreqSetGlobal;
         DoJsr(entryPoint);
      END;
END;
```

```
FUNCTION GetFieldByName(cardFieldFlag: BOOLEAN; fieldName: Str255): Handle;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(cardFieldFlag);
         inArgs[2] := ORD(@fieldName);
         request := xreqGetFieldByName;
         DoJsr(entryPoint);
         GetFieldByName := Handle(outArgs[1]);
      END;
END;


FUNCTION GetFieldByNum(cardFieldFlag: BOOLEAN; fieldNum: INTEGER): Handle;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(cardFieldFlag);
         inArgs[2] := fieldNum;
         request := xreqGetFieldByNum;
         DoJsr(entryPoint);
         GetFieldByNum := Handle(outArgs[1]);
      END;
END;


FUNCTION GetFieldByID(cardFieldFlag: BOOLEAN; fieldID: INTEGER): Handle;
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(cardFieldFlag);
         inArgs[2] := fieldID;
         request := xreqGetFieldByID;
         DoJsr(entryPoint);
         GetFieldByID := Handle(outArgs[1]);
      END;
END;
```

```
PROCEDURE SetFieldByName(cardFieldFlag: BOOLEAN; fieldName: Str255; fieldVal: Handle);
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(cardFieldFlag);
       inArgs[2] := ORD(@fieldName);
       inArgs[3] := ORD(fieldVal);
       request := xreqSetFieldByName;
       DoJsr(entryPoint);
     END;
END;


PROCEDURE SetFieldByNum(cardFieldFlag: BOOLEAN; fieldNum: INTEGER; fieldVal: Handle);
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(cardFieldFlag);
       inArgs[2] := fieldNum;
       inArgs[3] := ORD(fieldVal);
       request := xreqSetFieldByNum;
       DoJsr(entryPoint);
     END;
END;


PROCEDURE SetFieldByID(cardFieldFlag: BOOLEAN; fieldID: INTEGER; fieldVal: Handle);
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(cardFieldFlag);
       inArgs[2] := fieldID;
       inArgs[3] := ORD(fieldVal);
       request := xreqSetFieldByID;
       DoJsr(entryPoint);
     END;
END;


FUNCTION StringEqual(str1,str2: Str255): BOOLEAN;
BEGIN
   WITH paramPtr^ DO
     BEGIN
       inArgs[1] := ORD(@str1);
       inArgs[2] := ORD(@str2);
       request := xreqStringEqual;
       DoJsr(entryPoint);
       StringEqual := BOOLEAN(outArgs[1]);
     END;
END;
```

```
PROCEDURE ReturnToPas(zeroStr: Ptr; VAR pasStr: Str255);
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(zeroStr);
         inArgs[2] := ORD(@pasStr);
         request := xreqReturnToPas;
         DoJsr(entryPoint);
      END;
END;


PROCEDURE ScanToReturn(VAR scanPtr: Ptr);
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@scanPtr);
         request := xreqScanToReturn;
         DoJsr(entryPoint);
      END;
END;


PROCEDURE ScanToZero(VAR scanPtr: Ptr);
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(@scanPtr);
         request := xreqScanToZero;
         DoJsr(entryPoint);
      END;
END;


PROCEDURE ZeroBytes(dstPtr: Ptr; longCount: LongInt);
BEGIN
   WITH paramPtr^ DO
      BEGIN
         inArgs[1] := ORD(dstPtr);
         inArgs[2] := longCount;
         request := xreqZeroBytes;
         DoJsr(entryPoint);
      END;
END;
```

Glue routines in MPW C

The glue routines in MPW C follow:

```
/*
 *   XCmdGlue.c     - Implementation of HyperTalk callback routines
 *                  - Copyright Apple Computer, Inc. 1987,1988.
 *                  - All Rights Reserved.
 *
 *   #include "HyperXCmd.h" before your program.
 *   #include this file after your code.
 *
 */

pascal void SendCardMessage(paramPtr,msg)
     XCmdBlockPtr  paramPtr;      StringPtr      msg;
     /* Send a HyperCard message (a command with arguments) to the current card.
        msg is a pointer to a Pascal format string.  */
{
     paramPtr->inArgs[0] = (long)msg;
     paramPtr->request = xreqSendCardMessage;
     paramPtr->entryPoint();
}


pascal Handle EvalExpr(paramPtr,expr)
     XCmdBlockPtr paramPtr;       StringPtr      expr;
     /* Evaluate a HyperCard expression and return the answer.  The answer is
        a handle to a zero-terminated string. */
{
     paramPtr->inArgs[0] = (long)expr;
     paramPtr->request = xreqEvalExpr;
    paramPtr->entryPoint();
     return (Handle)paramPtr->outArgs[0];
}
```

```
pascal long StringLength(paramPtr,strPtr)
     XCmdBlockPtr paramPtr;       StringPtr     strPtr;
/* Count the characters from where strPtr points until the next zero byte.
   Does not count the zero itself.  strPtr must be a zero-terminated string. */
{
     paramPtr->inArgs[0] = (long)strPtr;
     paramPtr->request = xreqStringLength;
    paramPtr->entryPoint();
     return (long)paramPtr->outArgs[0];
}


pascal Ptr StringMatch(paramPtr,pattern,target)
     XCmdBlockPtr paramPtr;       StringPtr     pattern;      Ptr  target;
/* Perform case-insensitive match looking for pattern anywhere in
   target, returning a pointer to first character of the first match,
   in target or NIL if no match found.  pattern is a Pascal string,
   and target is a zero-terminated string. */
{
     paramPtr->inArgs[0] = (long)pattern;
     paramPtr->inArgs[1] = (long)target;
     paramPtr->request = xreqStringMatch;
    paramPtr->entryPoint();
     return (Ptr)paramPtr->outArgs[0];
}


pascal void SendHCMessage(paramPtr,msg)
     XCmdBlockPtr paramPtr;       StringPtr     msg;
     /* Send a HyperCard message (a command with arguments) to HyperCard.
        msg is a pointer to a Pascal format string.  */
{
     paramPtr->inArgs[0] = (long)msg;
     paramPtr->request = xreqSendHCMessage;
    paramPtr->entryPoint();
}
```

```
pascal void ZeroBytes(paramPtr,dstPtr,longCount)
     XCmdBlockPtr paramPtr;       Ptr  dstPtr;   long      longCount;
/* Write zeros into memory starting at destPtr and going for longCount
   number of bytes. */
{
     paramPtr->inArgs[0] = (long)dstPtr;
     paramPtr->inArgs[1] = longCount;
     paramPtr->request = xreqZeroBytes;
    paramPtr->entryPoint();
}


pascal Handle PasToZero(paramPtr,pasStr)
     XCmdBlockPtr  paramPtr;      StringPtr     pasStr;
/* Convert a Pascal string to a zero-terminated string.  Returns a handle
   to a new zero-terminated string.  The caller must dispose the handle.
   You'll need to do this for any result or argument you send from
   your XCMD to HyperTalk. */
{
     paramPtr->inArgs[0] = (long)pasStr;
     paramPtr->request = xreqPasToZero;
    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;
    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;
    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;
    paramPtr->entryPoint();
     return paramPtr->outArgs[0];
}


pascal Boolean StrToBool(paramPtr,str)
     XCmdBlockPtr  paramPtr;      Str31 *        str;
/* Convert the Pascal strings 'true' and 'false' to booleans. */
{
     paramPtr->inArgs[0] = (long)str;
     paramPtr->request = xreqStrToBool;
    paramPtr->entryPoint();
     return (Boolean)paramPtr->outArgs[0];
}


pascal void StrToExt(paramPtr,str,myext)
     XCmdBlockPtr  paramPtr;      Str31 *        str;      extended *     myext;
 /* Convert a string of ASCII decimal digits to an extended long integer.
     Instead of returning a new extended, as Pascal does, it expects you
     to create myext and pass it in to be filled. */
{
     paramPtr->inArgs[0] = (long)str;
     paramPtr->inArgs[1] = (long)myext;
     paramPtr->request = xreqStrToExt;
    paramPtr->entryPoint();
}
```

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


pascal void NumToHex(paramPtr,num,nDigits,mystr)
     XCmdBlockPtr  paramPtr;      long      num;
     short nDigits;      Str31 * mystr;
/* Convert an unsigned long integer to a hexadecimal number and put it
   into 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] = nDigits;
     paramPtr->inArgs[2] = (long)mystr;
     paramPtr->request = xreqNumToHex;
    paramPtr->entryPoint();
}
```

```
pascal void BoolToStr(paramPtr,bool,mystr)
     XCmdBlockPtr  paramPtr;      Boolean  bool;     Str31 *  mystr;
 /* Convert a boolean to 'true' or 'false'.  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)bool;
     paramPtr->inArgs[1] = (long)mystr;
     paramPtr->request = xreqBoolToStr;
    paramPtr->entryPoint();
}


pascal void ExtToStr(paramPtr,myext,mystr)
     XCmdBlockPtr  paramPtr;      extended *     myext;    Str31 *  mystr;
 /* Convert an extended long integer to decimal digits in a 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)myext;
     paramPtr->inArgs[1] = (long)mystr;
     paramPtr->request = xreqExtToStr;
    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;
    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;
    paramPtr->entryPoint();
}


pascal Handle GetFieldByName(paramPtr,cardFieldFlag,fieldName)
     XCmdBlockPtr  paramPtr;      Boolean   cardFieldFlag;
     StringPtr fieldName;
/* Return a handle to a zero-terminated string containing the value of
   field fieldName on the current card.  You must dispose the handle. */
{
     paramPtr->inArgs[0] = (long)cardFieldFlag;
     paramPtr->inArgs[1] = (long)fieldName;
     paramPtr->request = xreqGetFieldByName;
    paramPtr->entryPoint();
     return (Handle)paramPtr->outArgs[0];
}


pascal Handle GetFieldByNum(paramPtr,cardFieldFlag,fieldNum)
     XCmdBlockPtr  paramPtr;      Boolean   cardFieldFlag;
     short     fieldNum;
/* Return a handle to a zero-terminated string containing the value of
   field fieldNum on the current card.  You must dispose the handle. */
{
     paramPtr->inArgs[0] = (long)cardFieldFlag;
     paramPtr->inArgs[1] = fieldNum;
     paramPtr->request = xreqGetFieldByNum;
    paramPtr->entryPoint();
     return (Handle)paramPtr->outArgs[0];
}
```

```
pascal Handle GetFieldByID(paramPtr,cardFieldFlag,fieldID)
     XCmdBlockPtr  paramPtr;      Boolean  cardFieldFlag;
     short     fieldID;
/* Return a handle to a zero-terminated string containing the value of
   the field whise ID is fieldID.  You must dispose the handle. */
{
     paramPtr->inArgs[0] = (long)cardFieldFlag;
     paramPtr->inArgs[1] = fieldID;
     paramPtr->request = xreqGetFieldByID;
    paramPtr->entryPoint();
     return (Handle)paramPtr->outArgs[0];
}


pascal void SetFieldByName(paramPtr,cardFieldFlag,fieldName,fieldVal)
     XCmdBlockPtr  paramPtr;      Boolean  cardFieldFlag;
     StringPtr fieldName;    Handle    fieldVal;
/* Set the value of field fieldName to be the zero-terminated string
   in fieldVal.  The contents of the Handle are copied, so you must
   still dispose it afterwards. */
{
     paramPtr->inArgs[0] = (long)cardFieldFlag;
     paramPtr->inArgs[1] = (long)fieldName;
     paramPtr->inArgs[2] = (long)fieldVal;
     paramPtr->request = xreqSetFieldByName;
    paramPtr->entryPoint();
}


pascal void SetFieldByNum(paramPtr,cardFieldFlag,fieldNum,fieldVal)
     XCmdBlockPtr  paramPtr;      Boolean  cardFieldFlag;
     short     fieldNum;          Handle    fieldVal;
/* Set the value of field fieldNum to be the zero-terminated string
   in fieldVal.  The contents of the Handle are copied, so you must
   still dispose it afterwards. */
{
     paramPtr->inArgs[0] = (long)cardFieldFlag;
     paramPtr->inArgs[1] = fieldNum;
     paramPtr->inArgs[2] = (long)fieldVal;
     paramPtr->request = xreqSetFieldByNum;
    paramPtr->entryPoint();
}
```

```
pascal void SetFieldByID(paramPtr,cardFieldFlag,fieldID,fieldVal)
     XCmdBlockPtr  paramPtr;      Boolean   cardFieldFlag;
     short     fieldID;           Handle    fieldVal;
/* Set the value of the field whose ID is fieldID to be the zero-
   terminated string in fieldVal.  The contents of the Handle are
   copied, so you must still dispose it afterwards. */
{
     paramPtr->inArgs[0] = (long)cardFieldFlag;
     paramPtr->inArgs[1] = fieldID;
     paramPtr->inArgs[2] = (long)fieldVal;
     paramPtr->request = xreqSetFieldByID;
    paramPtr->entryPoint();
}


pascal Boolean StringEqual(paramPtr,str1,str2)
     XCmdBlockPtr  paramPtr;      Str31 *        str1;     Str31 *        str2;
/* Return true if the two strings have the same characters.
   Case insensitive compare of the strings. */
{
     paramPtr->inArgs[0] = (long)str1;
     paramPtr->inArgs[1] = (long)str2;
     paramPtr->request = xreqStringEqual;
    paramPtr->entryPoint();
     return (Boolean)paramPtr->outArgs[0];
}
```

```
pascal void ReturnToPas(paramPtr,zeroStr,pasStr)
     XCmdBlockPtr  paramPtr;      Ptr  zeroStr;      StringPtr    pasStr;
/* zeroStr points into a zero-terminated string.  Collect the
   characters from there to the next carriage Return and return
   them in the Pascal string pasStr.  If a Return is not found,
   collect chars until the end of the string. */
{
     paramPtr->inArgs[0] = (long)zeroStr;
     paramPtr->inArgs[1] = (long)pasStr;
     paramPtr->request = xreqReturnToPas;
    paramPtr->entryPoint();
}


pascal void ScanToReturn(paramPtr,scanHndl)
     XCmdBlockPtr  paramPtr;      Ptr *     scanHndl;
/* Move the pointer scanPtr along a zero-terminated
   string until it points at a Return character
   or a zero byte.  */
{
     paramPtr->inArgs[0] = (long)scanHndl;
     paramPtr->request = xreqScanToReturn;
    paramPtr->entryPoint();
}


pascal void ScanToZero(paramPtr,scanHndl)
     XCmdBlockPtr  paramPtr;      Ptr *     scanHndl;
/* Move the pointer scanPtr along a zero-terminated
   string until it points at a zero byte.  */
{
     paramPtr->inArgs[0] = (long)scanHndl;
     paramPtr->request = xreqScanToZero;
    paramPtr->entryPoint();
}
```

Attaching an XCMD or XFCN

To attach an existing XCMD or XFCN (one that has already been compiled or assembled into a resource) to one of your stacks, use a resource editor such as ResEdit. The following steps describe the procedure using ResEdit:

1. Launch ResEdit.
2. Select and open the stack containing the 'XCMD' or 'XFCN' resource you want.
3. Select and open the resource type of 'XCMD' or 'XFCN'.
4. Select and open the particular resource you want by name.
5. Press Command-C to copy the resource.
6. Select and open the stack you want to paste the resource into.
7. If your stack has no resource fork, ResEdit will display a dialog box asking if you want to open one. Click OK. ResEdit will open a window.
8. Press Command-V to paste the resource into your stack.
9. Click the close box on the window. When ResEdit asks if you want to save the file, click Yes.
10. Quit ResEdit.

HyperCard Developer's Toolkit

A disk containing the MPW Pascal and C definition and glue files described in this appendix is available from APDA, the Apple Programmer's and Developer's Association, exclusively to APDA members. You can order the disk and preliminary documentation in a package called the HyperCard Developer's Toolkit.

For membership and ordering information contact:

Apple Programmer's and Developer's Association
290 SW 43rd Street
Renton, WA 98055
Telephone: (206) 251-6548

Appendix B

ControlKey Parameters

This appendix lists the parameter variable values generated by HyperCard in response to different keys pressed in combination with the Control key.

When you press the Control key in combination with another key, HyperCard sends the system message `controlKey` to the current card with one integer parameter value:

`controlKey` *var*

The message can be intercepted by a handler placed anywhere in the object hierarchy between the current card and HyperCard. For example, the following handler causes the Control-P key combination to print the current card:

```
on controlKey whichKey
  if whichKey = 16 then
    doMenu "Print Card"
    exit controlKey
  end if
  pass controlKey
end controlKey
```

The `controlKey` system message is listed in Chapter 6.

Table B-1 lists the parameter values generated by various keys of the Apple Extended Keyboard pressed in combination with the Control key. Parameter values 1 through 31 represent American Standard Code for Information Interchange (ASCII) character code values for combinations of the Control key and letter keys. Some of the parameter values can be generated by more than one key. The parameter value is not affected by pressing the Shift key along with the Control key and the other key.

Table B-1
ControlKey message parameter values

Parameter value	Key(s)	Parameter value	Key(s)
1	a, Home	27	Esc, Clear, Left-bracket ([ )
2	b	28	Backslash (\), Left Arrow
3	c, Enter	29	Right bracket ( ]), Right Arrow
4	d, End	30	Up Arrow
5	e, Help	31	Hyphen (-), Down Arrow
6	f	39	Single Quotation Mark (')
7	g	42	Asterisk (*)
8	h, Delete	43	Plus (+)
9	i, Tab	44	Comma (,)
10	j	45	Minus (–)
11	k, Page Up	46	Period (.)
12	l, Page Down	47	Slash (/)
13	m, Return	48	0
14	n	49	1
15	o	50	2
16	p, all function keys	51	3
17	q	52	4
18	r	53	5
19	s	54	6
20	t	55	7
21	u	56	8
22	v	57	9
23	w	59	Semicolon (;)
24	x	61	Equal (=)
25	y	96	Tilde (~)
26	z	127	Forward Delete

Appendix C

Extended ASCII Table

This appendix lists the character assignments for the 256 single-byte character values used by Macintosh.

There are 256 possible 8-bit binary values, from 00000000 to 11111111. Of these, the first 128 (from 00000000 to 01111111) have been assigned to a standard set of characters and commands used in data processing and communication. These assignments form the ASCII character set. (*ASCII* stands for *American Standard Code for Information Interchange.*)

The remaining 128 binary values, those for which the most significant bit (first digit) is 1 instead of 0, are not assigned in the ASCII standard. Because they have higher numerical values that the first 128 characters, they are often referred to as high-ASCII characters.

This appendix lists all character values by their decimal equivalent.

Table C-1 lists the first 32 characters, the Control characters, which have no printable-character representation, with the standard abbreviation for each and its meaning.

Table C-1
Control character assignments

Value	Name	Meaning	Value	Name	Meaning
0	NUL	Null	16	DLE	Data link escape
1	SOH	Start of heading	17	DC1	Device control 1
2	STX	Start of text	18	DC2	Device control 2
3	ETX	End of text	19	DC3	Device control 3
4	EOT	End of transmission	20	DC4	Device control 4
5	ENQ	Enquiry	21	NAK	Negative acknowledge
6	ACK	Acknowledge	22	SYN	Synchronous idle
7	BEL	Bell	23	ETB	End of transmission block
8	BS	Backspace	24	CAN	Cancel
9	HT	Horizontal tab	25	EM	End of medium
10	LF	Line feed	26	SUB	Substitute
11	VT	Vertical tab	27	ESC	Escape
12	FF	Form feed	28	FS	File separator
13	CR	Carriage return	29	GS	Group separator
14	SO	Shift out	30	RS	Record separator
15	SI	Shift in	31	US	Unit separator

Table C-2 lists the remaining 224 character values with the characters to which they are assigned in the Macintosh Courier font.

Table C-2
Character assignments in Macintosh Courier font

Value	Character	Value	Character	Value	Character	Value	Character	Value	Character
32	SPACE	77	M	122	z	167	ß	212	`
33	!	78	N	123	{	168	®	213	'
34	"	79	O	124	|	169	©	214	+
35	#	80	P	125	}	170	™	215	◊
36	$	81	Q	126	~	171	´	216	ÿ
37	%	82	R	127	DEL	172	¨	217	Ÿ
38	&	83	S	128	Ä	173	≠	218	⁄
39	'	84	T	129	Å	174	Æ	219	¤
40	(	85	U	130	Ç	175	Ø	220	‹
41	)	86	V	131	É	176	∞	221	›
42	*	87	W	132	Ñ	177	±	222	fi
43	+	88	X	133	Ö	178	≤	223	fl
44	,	89	Y	134	Ü	179	≥	224	‡
45	-	90	Z	135	á	180	¥	225	·
46	.	91	[	136	à	181	µ	226	‚
47	/	92	\	137	â	182	∂	227	„
48	0	93	]	138	ä	183	∑	228	‰
49	1	94	^	139	ã	184	∏	229	Â
50	2	95	_	140	å	185	π	230	Ê
51	3	96	`	141	ç	186	∫	231	Á
52	4	97	a	142	é	187	ª	232	Ë
53	5	98	b	143	è	188	º	233	È
54	6	99	c	144	ê	189	Ω	234	Í
55	7	100	d	145	ë	190	æ	235	Î
56	8	101	e	146	í	191	ø	236	Ï
57	9	102	f	147	ì	192	¿	237	Ì
58	:	103	g	148	î	193	¡	238	Ó
59	;	104	h	149	ï	194	¬	239	Ô
60	<	105	i	150	ñ	195	√	240	
61	=	106	j	151	ó	196	ƒ	241	Ò
62	>	107	k	152	ò	197	≈	242	Ú
63	?	108	l	153	ô	198	Δ	243	Û
64	@	109	m	154	ö	199	«	244	Ù
65	A	110	n	155	õ	200	»	245	ı
66	B	111	o	156	ú	201	…	246	ˆ
67	C	112	p	157	ù	202	␣	247	˜
68	D	113	q	158	û	203	À	248	¯
69	E	114	r	159	ü	204	Ã	249	˘
70	F	115	s	160	†	205	Õ	250	˙
71	G	116	t	161	°	206	Œ	251	˚
72	H	117	u	162	¢	207	œ	252	¸
73	I	118	v	163	£	208	–	253	˝
74	J	119	w	164	§	209	—	254	˛
75	K	120	x	165	•	210	"	255	ˇ
76	L	121	y	166	¶	211	"		

␣ Stands for a nonbreaking space

Appendix D

Operator Precedence Table

This appendix shows the order of precedence of HyperTalk's operators. In a complex expression containing more than one operator, HyperCard performs the operation indicated by operators with lower-numbered precedence before those with higher-numbered precedence. Operators of equal precedence are evaluated left-to-right, except for exponentiation, which goes right-to-left. If you use parentheses, HyperCard evaluates the innermost parenthetical expression first.

Chapter 4 discusses expression evaluation.

Table D-1
Operator precedence

Order	Operators	Type of operator
1	( )	Grouping
2	-	Minus sign for numbers
	not	Logical negation for Boolean values
3	^	Exponentiation for numbers
4	* / div mod	Multiplication and division for numbers
5	+ -	Addition and subtraction for numbers
6	& &&	Concatenation of text
7	> < <= >= ≤ ≥	Comparison for numbers or text
	is in contains	Comparison for text
	is not in	Comparison for text
8	= is is not <> ≠	Comparison for numbers or text
9	and	Logical for Boolean values
10	or	Logical for Boolean values

Appendix E

HyperCard Limits

This appendix lists various minimum and maximum sizes and numbers of elements defined in HyperCard.

The maximum limits shown in this appendix are theoretical. Some of them are lower in practice. For example, HyperCard currently brings an entire card into memory at once, so the maximum size of a card is limited by available memory. It's possible that a card with a lot of text and long scripts, created while running HyperCard on a Macintosh with 2 megabytes of RAM, would not be able to be opened on a Macintosh with 1 megabyte. The current useful size of a card (or background) is therefore between 50 and 100 kilobytes.

The term *part*, in this appendix and internally in HyperCard, refers to buttons or fields. The value represented by `LongInt` is 2,147,483,647; the value represented by `Integer` is 32,767.

The figures listed in this appendix pertain to version 1.2 of HyperCard; some of them may change in future versions.

Table E-1
HyperCard limits

Item	Limit
Stack limits	
Stack size	512 megabytes
Minimum stack size	4896 bytes
Maximum total number of bitmaps, cards and backgrounds per stack	16,777,216
Maximum stack name size	31 characters
Maximum stack script size	30,000 characters

returnInField message, HyperCard sends a tabKey message to the field if the following conditions are true:

- returnInField is not intercepted by a handler
- the field is not a scrolling field
- the insertion point or selection is on the last line
- the field's autoTab property (described in this appendix) is true

Otherwise, HyperCard inserts a return character into the field. The tabKey message, if it's not intercepted, causes HyperCard to place the insertion point in the next field.

New and enhanced commands

HyperCard version 1.2 includes three new HyperTalk commands: lock screen, unlock screen, and select. In addition, three HyperTalk commands have been enhanced: the find command has two new options, and the hide and show commands can operate on the card or background picture.

Lock screen and unlock screen

The lock screen and unlock screen commands have the following syntax:

```
lock screen
unlock screen [with visualEffect]
```

VisualEffect is any of the forms of the visual command described in Chapter 7.

The lock screen command sets the lockScreen global property to true, preventing HyperCard from updating the screen. If you go to another card or do other actions that change the appearance of the screen, those changes are not displayed until the lockScreen property becomes false.

The unlock screen command sets the lockScreen property to false, allowing HyperCard to update the screen. In addition, the with *visualEffect* option specifies a single visual transition that occurs as the screen is updated.

Visual effects can't be compounded using unlock screen, as they can be using the visual command. Visual effects compounded by the visual command are not executed until a go command is encountered. HyperCard flushes unexecuted visual effects and sets lockScreen to false at idle time (in effect, at the end of all pending handlers).

Select

```
select objectDescriptor
select [preposition] chunkExpression of fieldDescriptor
select [preposition] text of fieldDescriptor
select empty
```

ObjectDescriptor is the descriptor of a button or field, or `me` or `target`; *preposition* is `before` or `after`; and *fieldDescriptor* is the descriptor of a field. (Button and field descriptors and the special descriptor `me` are explained in Chapter 3. The special descriptor `target` is explained in Chapter 2.)

The `select` *objectDescriptor* form chooses the appropriate tool and selects the object specified, as though you had chosen the tool and clicked the object manually with the mouse. The other forms select text in the specified field. `Before` and `after` can be used to place the insertion point relative to the specified text or chunk of text. Using a chunk expression without a preposition selects the entire chunk, highlighting the characters in the chunk. The `select empty` form deselects highlighted text or removes the insertion point. The following lines are examples of the `select` command:

```
select button 1 -- chooses button tool and selects card button 1
select before char 1 of field 2 -- places insertion point at start of field
select after text of field 2 -- places insertion point at end of field
select char 1 to 5 of card field 2 -- selects first five characters of field
```

Find

The new options for the `find` command are invoked by the following forms of syntax in addition to those shown in Chapter 7:

```
find whole expression [in field fieldDesignator]
find string expression [in field fieldDesignator]
```

Expression yields any string of characters, and *fieldDesignator* is a background field name, number, or ID number.

The `find whole` form (also invoked by pressing Shift-Command-F) lets you search for a specific word or phrase, including spaces. For HyperCard to find a match, all the characters must be in the same field, and they must be in the same consecutive order as they appear in the string derived from *expression*.

In the following example, *expression* is a literal, yielding the string of characters between the double quotation marks:

```
find whole "Apple Computer"
```

The example finds a card with a field that has the phrase *Apple Computer* in it; it won't find *Apple Computers* or *This apple is a computer.* (The `find` command without `whole` would find a match in all three cases.) `Find whole` won't find partial-word matches, and it pays no attention to case or diacritical marks: *apple Cømpüter* and *aPPle cOmputer* are seen as the same.

When you use `find` without `whole`, HyperCard finds a card that contains every word in the string derived from *expression*, but the words can appear in different order or in different fields. That is, with `find whole`, interword spaces are part of the search string; without `whole` the spaces delimit separate search strings. With every form of `find`, you can limit the search to a specific background field.

The `find string` form lets you search for a contiguous string of characters, including spaces, regardless of word boundaries. (`Find whole` searches for characters at the beginnings of words.) For HyperCard to find a match, all the characters must be in the same field, and they must be in the same order as in the string derived from *expression*. For strings without spaces, `find string` works the same as `find chars`.

In this example:

```
find string "ple Computer"
```

HyperCard finds the string in *Apple computers* but not in *computers, not apples.* (The `find` command without `string` would not find a match in either case.)

Hide and show

The `hide` and `show` commands in version 1.2 operate on the bitmap pictures on cards and backgrounds, as well as the menu bar, card window, Message box, Tools and Patterns palettes, and buttons and fields, as described in Chapter 7. The syntax for the new forms is:

```
hide card picture
hide picture of cardDescriptor

hide background picture
hide picture of backgroundDescriptor

show card picture
show picture of cardDescriptor

show background picture
show picture of backgroundDescriptor
```

CardDescriptor yields the descriptor of a card in the current stack, and *backgroundDescriptor* yields the descriptor of a background in the current stack, as described in Chapter 3, "Naming Objects."

The `picture` form of the `hide` command removes from view the graphic bitmap on the card or background, and the `picture` form of the `show` command displays it.

Hidden card and background pictures are not displayed when the Browse, Button, or Field tools are chosen, but if you attempt to use a Paint tool manually, a dialog box appears asking if you want to make the picture visible; clicking OK displays the picture. (You can draw on hidden pictures from a script.) Whether or not you are in Edit background mode determines whether your actions pertain to the card or background picture.

The following example,

```
show picture of card 3
```

makes the graphic bitmap of the third card in the current stack visible, setting the card's `showPict` property to `true`. If the picture were visible before you issued the `show picture` command, of course, there would be no effect.

New and enhanced functions

HyperCard version 1.2 has two enhancements to existing functions and nine new functions.

Enhancement to number function

The `number` function has been enhanced by the following form:

`the number of cards of` *backgroundDescriptor*

BackgroundDescriptor yields the descriptor of a background in the current stack, as described in Chapter 3, "Naming Objects."

This form of the `number` function returns the number of cards that are associated with the specified background. For example,

```
get the number of cards of background 3
```

Enhancement to version function

The `version` function has been enhanced by the two following forms:

```
the [long] version [of HyperCard]
the version of stackDescriptor
```

StackDescriptor yields the descriptor of any stack currently available to your Macintosh, as described in Chapter 3, "Naming Objects."

The `long` modifier, when used with the `version [of HyperCard]` form, returns the standard Macintosh version resource format (see *Inside Macintosh* for details). For example,

```
put the long version -- returns 01208000 for HyperCard version 1.2
```

The `version of` *stackDescriptor* form returns a five-item string including the following items:

- the version of HyperCard that created the stack
- the version of HyperCard last used to compact the stack
- the oldest version of HyperCard that changed the stack since it was last compacted
- the version of HyperCard that last changed the stack
- the date when the stack was most recently modified and closed (in seconds since midnight, January 1, 1904)

All versions are returned in the `long` form, and the version numbers are separated by commas. Items 1 through 4 are set to 00000000 if the version of HyperCard is less than 1.2. For example, if `old stack` were created with HyperCard version 1.1, then edited and compacted with version 1.2,

```
put the version of stack "old stack"
```

would put the following value in the Message box:

```
00000000,01208000,01208000,01208000,2660687462
```

You can use the `convert` command, described in Chapter 7, to change item 5 into a more readable format.

Functions for found text

```
the foundText
the foundChunk
the foundLine
the foundField
```

These functions return information about text found by the `find` command. The `foundText` function returns the characters that are enclosed in the box after the `find` command has executed successfully; for example, the commands

```
find "Hyper"
put the foundText
```

would put `HyperCard` in the Message box if it were the word containing the matching string. The `foundChunk` function returns a chunk expression describing the location of the text in the box; for example, if field 1 contained *Now is the time,* the commands

```
find "Now"
put the foundChunk
```

would put `char 1 to 3 of bkgnd field 1` into the Message box. The `foundLine` function returns a chunk expression describing the line in which the beginning of the text was found, in a form such as `line 1 of card field 2`. The `foundField` function returns the descriptor of the field in which the text was found, in a form such as `card field 2`.

Functions for selected text

```
the selectedText
the selectedChunk
the selectedLine
the selectedField
```

These functions return information about text that is currently selected. The `selectedText` function returns the selected text itself. The `selectedChunk` function returns a chunk expression describing the location of the selected text, the `selectedLine` returns a chunk expression describing the line containing the selected text, and the `selectedField` returns the descriptor of the field containing the selected text. The forms of the expressions returned by these functions are like those returned by the functions for found text, described in the previous section.

ScreenRect function

```
the screenRect
screenRect()
```

The `screenRect` function returns the rectangle of the screen in which HyperCard's menu bar is displayed; the value returned is four integers, separated by commas, representing the pixel offsets of the left, top, right, and bottom edges, respectively, from the top-left corner of the screen.

See also "Properties of screen rectangles," in the next section.

New and enhanced properties

HyperCard version 1.2 has five new HyperTalk properties: `autoTab`, `cantDelete`, `cantModify`, `showPict`, and `userModify`. All five properties can have values of `true` or `false`. In addition, version 1.2 has an enhanced `cursor` global property and eight new ways to specify aspects of the screen rectangles of buttons, fields, and windows.

AutoTab

```
set autoTab of field 3 to true
```

The `autoTab` property pertains to any nonscrolling field in the current stack. When `autoTab` is `true`, pressing Return with the insertion point in the last line of that field moves the insertion point to the next field on that card by sending the `tabKey` message to the current card.

(Normal tabbing order is followed: if the field you're leaving is a card field, the insertion point moves to the next higher-numbered card field or to the lowest-numbered background field if no higher-numbered card field exists; if the field you're leaving is a background field, the insertion point moves to the next higher-numbered background field or to the lowest-numbered card field if no higher-numbered background field exists.)

The `autoTab` property can also be set by clicking the Auto Tab check box in the Field Info dialog box of the nonscrolling field.

CantDelete

```
set cantDelete of this card to true
```

The `cantDelete` property pertains to any card or background in the current stack, or to any stack accessible to your Macintosh. It controls whether or not the user can delete the specified card, background, or stack. This property checks or unchecks the "Can't delete" option in the object Info dialog box of the specified object.

The `cantDelete` property is also automatically set when the user sets `cantModify`, as described in the following section.

CantModify

```
set cantModify of this stack to true
```

The `cantModify` property pertains to any stack accessible to your Macintosh. It controls whether or not the stack can be changed in any way. This property checks or unchecks both the "Can't modify" stack option and the "Can't delete stack" option in the Protect Stack dialog box. (If the user has checked "Can't delete stack," however, and a script sets `cantModify` to `true` and then `false`, "Can't delete stack" is left checked.)

When you set `cantModify` from a script, you override whatever the user has set by hand in the Protect Stack dialog box. Setting `cantModify` to `false` does not, however, override protection provided by media that are write-protected in other ways.

See also the `userModify` property, later in this appendix.

ShowPict

```
set showPict of this card to false
```

The `showPict` property pertains to a card or a background in the current stack. It controls whether or not the specified card or background picture is displayed. Setting the `showPict` property of a card or background to `false` is the same as hiding it with the `picture` form of the `hide` command, described in this appendix; setting it to `true` is the same as showing it with the `picture` form of the `show` command.

When the `showPict` property of the current card or background is `false` and you attempt to use a Paint tool on it manually, a dialog box appears asking if you want to make the picture visible; clicking OK sets the `showPict` property to `true` and the picture appears. (You can draw on hidden pictures from a script.)

UserModify

```
set userModify to true
```

The `userModify` property is a global property pertaining to HyperCard itself. It controls whether or not a user can type into fields or use Paint tools on a card that has been write-protected. A card is write-protected under the following circumstances:

- The stack is on a CD-ROM.
- The stack is on a file server in a folder whose access privileges are set to Read Only.
- The "Locked" box is checked in the stack's Get Info dialog box in the Finder's File menu.
- The stack is on a locked 3.5-inch disk.
- "Can't modify stack" is checked in the stack's Protect Stack dialog box.

Cursor

```
set cursor to busy
```

The `cursor` property has been enhanced to accept eight cursor names by default: `arrow`, `busy`, `cross`, `hand`, `iBeam`, `none`, `plus`, and `watch`. You can also set the cursor to the ID number or name of any available 'CURS' resource, as explained in Chapter 9. The `busy` cursor is HyperCard's beach ball—each time it's set, it turns 45° clockwise, so you can make it appear to spin by setting it inside a `repeat` loop:

```
on mouseUp
  repeat until the mouseClick
    set cursor to busy
    wait 2 ticks
  end repeat
end mouseUp
```

Properties of screen rectangles

The properties described in this section pertain to the screen rectangles of buttons and fields, the Tools and Patterns palettes, the Message box, and the card window.

`the left of` *partOrWindow*

ParOrWindow yields the descriptor of a button or field in the current stack, as described in Chapter 3, "Naming Objects," or the name of one of the windows listed above, as described in Chapter 9, "Properties."

You use the `left` property to determine or change item 1 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

`the top of` *partOrWindow*

You use the `top` property to determine or change item 2 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

`the right of` *partOrWindow*

You use the `right` property to determine or change item 3 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

`the bottom of` *partOrWindow*

You use the `bottom` property to determine or change item 4 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

`the topLeft of` *partOrWindow*

You use the `topLeft` property to determine or change items 1 and 2 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window.

`the bottomRight of` *partOrWindow*

You use the `bottomRight` property to determine or change items 3 and 4 of the value of the `rectangle` property (left, top, right, bottom) when applied to the specified object or window. The `bottomRight` property can be abbreviated `botRight`.

`the width of` *partOrWindow*

You use the `width` property to determine the horizontal distance in pixels occupied by the rectangle of the specified object or window. You can change the width of a button or field rectangle with the `set` command, but you can't set that property of a window.

`the height of` *partOrWindow*

You use the `height` property to determine or change the vertical distance in pixels occupied by the rectangle of the specified object or window. You can change the height of a button or field rectangle with the `set` command, but you can't set that property of a window.

When you set the width or height of a button or field, its `location` property (center coordinate) remains the same.

New operator

HyperCard version 1.2 has one new operator: `within`. It pertains to the screen rectangles of buttons and fields, the Tools and Patterns palettes, the Message box, and the screen on which HyperCard's menu bar is displayed. The syntax of an expression in which `within` is valid is the following:

point `is [not] within` *rectangle*

Point is an expression that yields a list of two integers separated by a comma and *rectangle* is an expression that yields a list of four integers separated by commas.

The `within` operator tests whether or not a point lies inside a rectangle; it results in a Boolean value: `true` or `false`. The following example handler, placed in a button script, is invoked when you click the button. It waits until you move the pointer outside the button rectangle, then beeps when you move the pointer back inside the button rectangle:

```
on mouseUp
  wait until the mouseLoc is not within rect of me
  repeat until the mouseLoc is within rect of me
    set cursor to busy -- spin beach ball while we wait
  end repeat
  beep
end mouseUp
```

New synonyms

HyperCard version 1.2 has twelve new synonyms (or abbreviations) for HyperTalk terms, which are shown in Table F-2. (The new abbreviations are additions, not replacements for the older terms.)

Table F-2
New HyperTalk synonyms

New synonym	Term
bg	background
bgs	backgrounds
btns	buttons
cd	card
cds	cards
fld	field
flds	fields
grey	gray
pict	picture
sec	secs or seconds
second	secs or seconds
tick	ticks

New shortcuts

HyperCard version 1.2 has several new keyboard shortcuts that allow you to edit scripts of objects more easily.

Command-Tab

In all versions of HyperCard, pressing Command-Tab chooses the Browse tool. In version 1.2, two additional shortcuts are available: holding down the Command key and pressing Tab twice in rapid succession chooses the Button tool; holding down the Command key and pressing Tab three times in rapid succession chooses the Field tool. The period of time defining *rapid succession* is 30 ticks (one-half second).

Command-Option

While using the Browse tool, you can press the Command and Option keys simultaneously to display the outline of all visible buttons (those whose visible property is `true`). While the buttons are displayed this way, you can click one to edit its script.

While using the Button tool, you can use the Command-Option combination to display all buttons (visible and hidden). However, the click-to-edit shortcut works for the visible buttons only. The user level must be set to Scripting to edit scripts.

Shift-Command-Option

While using the Browse tool, you can press the Shift, Command, and Option keys simultaneously to display the outline of all visible fields (those whose visible property is `true`). While the fields are displayed this way, you can click one to edit its script.

While using the Field tool, you can use the Shift-Command-Option combination to display all fields (visible and hidden). However, the click-to-edit shortcut works for the visible fields only. The user level must be set to Scripting to edit scripts.

Other Command-Option key combinations

When you're using any tool, Command-Option-C edits (invokes the script editor for) the script of the current card, Command-Option-B edits the script of current background, and Command-Option-S edits the script of the current stack.

The shortcuts introduced with HyperCard version 1.2 are summarized in Table F-3.

Table F-3
New shortcuts

Key press	Effect
Command-Tab	Choose Browse tool
Command-Tab(2x)	Choose Button tool
Command-Tab(3x)	Choose Field tool
Command-Option	Display buttons; click to edit script
Shift-Command-Option	Display fields; click to edit script
Command-Option-C	Edit script of current card
Command-Option-B	Edit script of current background
Command-Option-S	Edit script of current stack

Appendix G

HyperTalk Syntax Summary

This appendix lists HyperTalk's built-in commands and functions, showing the syntax of their parameters.

HyperTalk's built-in commands and functions are described in more detail in Chapters 7 and 8, respectively. A brief description and page reference for each is included in Appendix G.

Syntax description notation

The syntax descriptions use the following typographic conventions. Words or phrases in `typewriter type` are Hypertalk language elements or are those that you type to the computer literally, exactly as shown. Words in *italic* type describe general elements, not specific names—you must substitute the actual instances. Square brackets ([ ]) enclose optional elements which may be included if you need them. (Don't type the square brackets.) In some cases, optional elements change what the message does; in other cases they are helper words that have no effect except to make the message more readable.

It doesn't matter whether you use uppercase or lowercase letters; names that are formed from two words are shown in small letters with a capital in the middle (`likeThis`) merely to make them more readable. The HyperTalk prepositions `of` and `in` are interchangeable—the syntax descriptions use the one that sounds more natural.

The terms *factor* and *expression* are defined in Chapter 4. Briefly, a factor can be a constant, literal, function, property, number, or container, and an expression can be a factor or a complex expression built with factors and operators. Also, a factor can be an expression within parentheses.

Table G-1
HyperTalk command syntax

```
add expression to destination
answer question [with reply [or reply2 [or reply3]]]
arrowKey keyName
ask [password] question [with defaultAnswer]
beep count
choose toolName tool
click at location [with key[,key2[,key3]]]
close file fileName
close printing
convert container to format [and format]
delete chunk [of container]
dial expression [with modem [modemCommands]]
divide destination by expression
doMenu menuItem
drag from start to finish [with key[,key2[,key3]]]
edit script of object
enterKey
find [chars] expression [in field fieldDesignator]
find [word] expression [in field fieldDesignator]
functionKey keyNumber
get expression
go [to] [stack] stackName
go [to] bkgndDescriptor [of [stack] stackName]
go [to] cardDescriptor [of bkgndDescriptor] [of [stack] stackName]
help
hide menuBar
hide window
hide part
multiply destination by expression
open [document with] application
open file fileName
open printing [with dialog]
play "voice" [tempo] ["notes"]
play stop
pop card [preposition destination]
print card
print expression cards
print cardDescriptor
print document with application
push cardDescriptor
put expression [preposition destination]
read from file fileName until character
read from file fileName for numberOfCharacters
```

```
reset paint
returnKey
set [the] property [of object] to value
show number cards
show menuBar
show window [at h, v]
show part [at h, v]
sort [direction] [style] by expression
subtract expression from destination
tabKey
type expression [with key[, key2[, key3]]]
visual [effect] effectName [speed] [to image]
wait [for] time [seconds]
wait until condition
wait while condition
write source to file fileName
```

Table G-2
HyperTalk function syntax

```
the abs of factor
abs(expression)
annuity(rate, periods)
the atan of factor
atan(expression)
average(list)
the charToNum of factor
charToNum(expression)
the clickLoc
clickLoc()
the commandKey
commandKey()
compound(rate, periods)
the cos of factor
cos(expression)
the [modifier] date
the diskSpace
diskSpace()
the exp of factor
exp(expression)
the exp1 of factor
exp1(expression)
the exp2 of factor
exp2(expression)
the length of factor
```

```
length(expression)
the ln of factor
ln(expression)
the ln1 of factor
ln1(expression)
the log2 of factor
log2(expression)
max(list)
min(list)
the mouse
mouse()
the mouseClick
mouseClick()
the mouseH
mouseH()
the mouseLoc
mouseLoc()
the mouseV
mouseV()
[the] number of objects
[the] number of chunks in factor
the numToChar of factor
numToChar(expression)
offset(string1, string2)
the optionKey
optionKey()
the param of factor
param(expression)
the paramCount
paramCount()
the params
params()
the random of factor
random(expression)
the result
result()
the round of factor
round(expression)
the seconds
seconds()
the shiftKey
shiftKey()
the sin of factor
sin(expression)
the sound
```

```
sound()
the sqrt of factor
sqrt(expression)
the tan of factor
tan(expression)
the target
target()
the ticks
ticks()
the [adjective] time
time()
the tool
tool()
the trunc of factor
trunc(expression)
the value of factor
value(expression)
the version
version()
```

Appendix H

HyperTalk Vocabulary

This appendix lists, in alphabetical order, HyperTalk's native vocabulary—the names of its built-in commands and functions, its system messages, keywords, the names of objects and their properties, and various adjectives, constants, ordinals, and other terms.

This list is not exhaustive—there are other terms with specific meanings recognized by HyperCard in particular contexts, and they are described with the primary term to which they relate. For example, the names of the various visual effects are listed with the `visual` command in Chapter 7.

The parameter syntax of HyperTalk's built-in commands and functions is shown in Appendix G.

Table H-1
HyperTalk vocabulary

Term	Category	Page	Meaning
`abbr[ev[iated]]`	Adjective	145, 175	Modifies the value returned by the `date` function or the `name` or `ID` properties.
`abs`	Function	140	Returns absolute value of a number.
`add`	Command	88	Adds the value of an expression to a value in a container.
`after`	Preposition	122	Used with `put` command, directing HyperCard to append a new value following any preexisting value in a container.
`all`	Adjective	127	Specifies total number of cards in stack to `show cards` command.
`annuity`	Function	140	Computes present or future value of an ordinary annuity.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
answer	Command	89	Displays a dialog with question and reply buttons.
any	Ordinal	37	Special ordinal used with object or chunk to specify a random element within its enclosing set.
arrowKey	Command	90	Takes you to another card.
arrowKey	System message	82	Sent to current card when an arrow key is pressed.
ask	Command	92	Displays a dialog box with a question and default answer.
atan	Function	141	Returns trigonometric arc tangent of a number.
autoHilite	Property	203	Determines whether or not the specified button's `hilite` property is affected by the message `mouseDown`.
average	Function	142	Returns the average value of numbers in a list.
background	Object	3, 34	Generic name of background object; used with specific designation (`go to next background`). Also used to specify containing object for buttons and, optionally, fields (`background button 2`).
backgrounds	Object type	154	Specifies backgrounds as type of object to the `number` function.
beep	Command	93	Causes Macintosh to make a beep sound.
before	Preposition	122	Used with `put` command, directing HyperCard to place a new value at the beginning of any preexisting value in a container.
bkgnd	Object	34	Abbreviation for `background`.
bkgnds	Object type	155	Specifies backgrounds as type of object to the `number` function.
blindTyping	Property	176	Allows typing into Message box when hidden.
browse	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
brush	Property	184	Determines the current brush shape.
brush	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
btn	Object	34	Abbreviation for `button`.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
bucket	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
button	Object	34	Generic name of button object; used with a specific designation (`hide button one`).
button	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
buttons	Object type	154	Specifies buttons as type of object to the `number` function.
card	Object	34	Generic name of a card object; used with a specific designation (`go to card "fred"`). Also used to specify containing object for fields and, optionally, buttons (`card field "date"`).
cards	Object type	154	Specifies cards as type of object to the `number` function.
centered	Property	184	Determines the Draw Centered setting.
char[acter]	Chunk	54	A character of text in any container or expression.
char[acter]s	Chunk type	154	Specifies characters as type of chunk to the `number` function.
charToNum	Function	142	Returns ASCII value of a character.
choose	Command	94	Changes the current tool.
click	Command	95	Causes same actions as clicking at a specified location.
clickLoc	Function	143	Returns location of most recent click.
closeBackground	System message	183	Sent to current card just before you leave the current background.
closeCard	System message	80	Sent to current card just before you leave it.
closeField	System message	79	Sent to unlocked field when it closes.
closeStack	System message	83	Sent to current card just before you leave the current stack.
close file	Command	97	Closes a previously opened disk file.
close printing	Command	98	Ends a print job.
commandKey	Function	143	Returns state of the Command key: `up` or `down`.
compound	Function	144	Computes present or future value of a compound interest–bearing account.
controlKey	System message	82	Sent to current card when a combination of the Control key and another key is pressed.
convert	Command	98	Converts a date or time to specified format.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
cos	Function	145	Returns the cosine of the angle that is passed to it.
cursor	Property	177	Sets image appearing at pointer location on screen. You can only set cursor; you can't get it.
curve	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
date	Function	145	Returns a string representing the current date.
delete	Command	100	Removes a chunk of text from a container.
deleteBackground	System message	83	Sent to current card just before the background is deleted.
deleteButton	System message	77	Sent to a button just before it is deleted.
deleteCard	System message	80	Sent to current card just before it is deleted.
deleteField	System message	79	Sent to a field just before it is deleted.
deleteStack	System message	83	Sent to the current card just before a stack is deleted.
dial	Command	101	Generates touch-tone sounds through audio output or modem attached to serial port.
diskSpace	Function	146	Displays the amount of free space available on the disk containing the current stack.
divide	Command	102	Divides the value in a container by the value of an expression.
do	Keyword	72	Sends the value of an expression as a message to the current card.
doMenu	Command	103	Performs a specified menu command.
doMenu	System message	83	Sent to current card when any menu item is chosen.
down	Constant	213	Value returned by various functions to describe the state of a key or the mouse button.
drag	Command	104	Performs same action as a manual drag.
dragSpeed	Property	177	Sets pixels-per-second speed at which pointer moves with `drag` command.
editBkgnd	Property	177	Determines whether manipulation of buttons, fields or paintings occurs on current card or background.
edit script	Command	105	Opens the script of a specified object.
eight	Constant	213	String representation of the numerical value 8.
eighth	Ordinal	36	Designates object or chunk number eight within its enclosing set.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
`else`	Keyword	70	Optionally follows `then` clause in an `if` structure to introduce an alternative action clause.
`empty`	Constant	213	The null string; same as the literal `""`.
`end`	Keyword	61, 64, 70, 71	Marks the end of a message handler, function handler, `repeat` loop, or multiple-statement `then` or `else` clause of an `if` structure.
`enterKey`	Command	105	Sends contents of Message box to the current card.
`eraser`	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`exit`	Keyword	61, 64, 69	Immediately ends execution of a message handler, function handler, or `repeat` loop.
`exp`	Function	147	Returns the mathematical exponential of its argument.
`exp1`	Function	147	Returns one less than the mathematical exponential of its argument.
`exp2`	Function	148	Returns the value of 2 raised to the power specified by the argument.
`false`	Constant	213	Boolean value resulting from evaluation of a comparative expression and returned from some functions.
`field`	Container	45	Generic name of field container; used with specific designation (`put the time into card field "time"`).
`field`	Object	2, 34	Generic name of field object; used with specific designation (`get name of first field`).
`field`	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`fields`	Object type	154	Specifies fields as type of object to the `number` function.
`fifth`	Ordinal	36	Designates object or chunk number five within its enclosing set.
`filled`	Property	185	Determines the Draw Filled setting.
`find`	Command	106	Searches card and background fields for text strings derived from an expression.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
first	Ordinal	36	Designates object or chunk number one within its enclosing set.
five	Constant	213	String representation of the numerical value 5.
formFeed	Constant	213	The form feed character (ASCII 12), which starts a new page in some file formats.
four	Constant	213	String representation of the numerical value 4.
fourth	Ordinal	36	Designates object or chunk number four within its enclosing set.
freeSize	Property	190	Determines the amount of free space available in a specified stack.
functionKey	Command	108	Performs Undo, Cut, Copy, or Paste operations with parameter values of 1, 2, 3, or 4, respectively.
functionKey	System message	82	Sent to current card when any function key on the Apple Extended Keyboard is pressed.
get	Command	109	Puts the value of an expression into the local variable `It`.
global	Keyword	73	Declares specified variables to be valid beyond current execution of current handler.
go	Command	110	Takes you to a specified card or stack.
grid	Property	185	Determines the Grid setting.
help	Command	111	Takes you to the first card in the stack named Help.
hide	Command	111	Hides the specified window from view.
hilite	Property	204	Determines whether a specified button is highlighted.
icon	Property	204	Determines the icon that is displayed with a specified button.
ID	Property	35, 192	Determines the permanent ID number of a specified background, card, field, or button. (See also pages 193, 195, and 205.)
idle	System message	81	Sent to the current card repeatedly whenever nothing else is happening.
if	Keyword	70	Introduces a conditional structure containing statements to be executed only if a specified condition is true.
into	Preposition	122	Used with `put` command, directing HyperCard to replace any preexisting value in a container with a new value.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
`It`	Container	47	Local variable that is the default destination for `get`, `ask`, `answer`, `read`, and `convert` commands.
`item`	Chunk	55	A piece of text delimited by commas in any container or expression.
`items`	Chunk type	154	Specifies items as type of chunk to the `number` function.
`language`	Property	178	Used to choose language in which scripts are displayed.
`lasso`	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`last`	Ordinal	37	Special ordinal used with object or chunk to specify the element whose number is equal to the total number of elements in its enclosing set.
`length`	Function	148	Returns the number of characters in the text string derived from an expression.
`line`	Chunk	55	A piece of text delimited by return characters in any container.
`line`	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`lineFeed`	Constant	213	The line feed character (ASCII 10), which starts a new line in some file formats.
`lines`	Chunk type	154	Specifies lines as type of chunk to the `number` function.
`lineSize`	Property	185	Determines the thickness of lines drawn with line and shape tools.
`ln`	Function	149	Returns the base-*e* (natural logarithm) of the number passed to it.
`ln1`	Function	149	Returns the base-*e* (natural logarithm) of the sum of the number passed to it plus 1.
`loc[ation]`	Property	182	Determines the location at which a window, field, or button is displayed. (See also pages 196 and 205.)
`lockMessages`	Property	178	Prevents HyperCard from sending all automatic messages such as `openCard`.
`lockRecent`	Property	178	Prevents HyperCard from adding miniature representations to the Recent card.
`lockScreen`	Property	179	Prevents updating of the screen from card to card.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
`lockText`	Property	196	Allows or prevents text editing in a specified field.
`log2`	Function	150	Returns the base-2 logarithm of the number passed to it.
`long`	Adjective	145, 175	Modifies value returned by `date` function and by `name` and `ID` properties.
`max`	Function	150	Returns the highest-value number from a list of numbers.
`me`	Object	37	Specifies object containing the executing handler.
`message [box]`	Container	48	The Message box.
`mid[dle]`	Ordinal	37	Special ordinal used with object or chunk to specify the element whose number is equal to one more than half the total number of elements in its enclosing set.
`min`	Function	151	Returns the lowest-value number from a list of numbers.
`mouse`	Function	151	Returns state of the mouse button: `up` or `down`.
`mouseClick`	Function	152	Determines whether the mouse button has been clicked.
`mouseDown`	System message	77, 79, 80	Sent to a button, unlocked field, or the current card when the mouse button is pressed down.
`mouseEnter`	System message	78, 79	Sent to a button or field when the pointer is first moved inside its rectangle.
`mouseH`	Function	153	Returns the horizontal offset in pixels of the pointer from the left edge of the card window.
`mouseLeave`	System message	78, 79	Sent to a button or field when the pointer is first removed from its rectangle.
`mouseLoc`	Function	153	Returns the point on the screen where the pointer is currently located.
`mouseStillDown`	System message	77, 79, 80	Sent to a button, unlocked field, or the current card repeatedly when the mouse button is held down.
`mouseUp`	System message	77, 79, 80	Sent to a button, unlocked field, or the current card when the mouse button is released after having been previously pressed down within the same object's rectangle.
`mouseV`	Function	154	Returns the vertical offset in pixels of the pointer from the top of the screen.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
`polySides`	Property	187	Determines the number of sides created by the Regular Polygon tool.
`pop card`	Command	118	Returns you to last card saved with the push card Command.
`powerKeys`	Property	180	Keyboard equivalents of commonly used painting actions.
`prev[ious]`	Object modifier	37	Used with `card` or `background` to refer to the one preceding the current one.
`print card`	Command	119	Prints the the current card or a specified number of cards beginning with the current card.
`print`	Command	120	Prints the specified document.
`push`	Command	121	Saves the identification of a specified card in a LIFO memory stack for later retrieval.
`put`	Command	122	Copies the value of an expression into a container.
`quit`	System message	84	Sent to the current card when you choose Quit HyperCard from the File menu (or press Command-Q), just before HyperCard goes away.
`quote`	Constant	213	The straight double quotation mark character.
`random`	Function	160	Returns a random integer between 1 and the integer derived from a specified expression.
`read`	Command	123	Reads a file previously opened with the `open file` command into the local variable `It`.
`rect[angle]`	Property	182	Determines the rectangle occupied by a specified window, field, or button. (See also pages 197 and 206.)
`rect[angle]`	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`reg[ular] poly[gon]`	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
`repeat`	Keyword	66	Introduces a `repeat` loop, an iterative structure containing a block of one or more statements executed multiple times.
`reset paint`	Command	125	Reinstates the default values of all the painting properties.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
result	Function	161	Returns the status of `find` or `go` command previously executed in current handler.
resume	System message	84	Sent to the current card when HyperCard resumes running after having been suspended.
return	Keyword	62, 65	Returns a value from a function handler or message handler.
returnKey	Command	125	Sends any statement in the Message box to the current card.
returnKey	System message	81	Sent to current card when Return key is pressed.
round	Function	163	Returns the number derived from an expression, rounded off to the nearest integer.
round rect [angle]	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
script	Property	191	Retrieves or replaces the script of the specified stack, background, card, field, or button. (See also pages 193, 194, 198, and 207.)
scroll	Property	198	Determines the amount of material that is hidden above the top of the specified scrolling field's rectangle.
second	Ordinal	36	Designates object or chunk number two within its enclosing set.
seconds	Function	163	Returns the number of seconds between midnight, January 1, 1904, and the current time.
select	Tool	94, 170	Name of tool from Tools palette; used with `choose` command or returned by the `tool` function.
selection	Container	47	Currently selected area of text in a field.
send	Keyword	74	Sends a specified message directly to a specified object.
set	Command	126	Changes the state of a specified global, painting, window, or object property.
seven	Constant	213	String representation of the numerical value 7.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
seventh	Ordinal	36	Designates object or chunk number seven within its enclosing set.
shiftKey	Function	164	Returns the state of the Shift key: up or down.
short	Adjective	145, 175	Modifies value returned by date function and by name and ID properties.
show cards	Command	127	Displays a specified number of cards in the current stack.
showLines	Property	199	Determines whether or not the text baselines are visible in a field.
showName	Property	207	Determines whether or not the name of a specified button is displayed in its rectangle on the screen.
show	Command	128	Displays a specified window or object.
sin	Function	164	Returns the sine of the angle that is passed to it.
six	Constant	213	String representation of the numerical value 6.
sixth	Ordinal	36	Designates object or chunk number six within its enclosing set.
size	Property	191	Returns the size of a specified stack.
sort	Command	130	Puts all of the cards in a specified stack in order, according to a specified key expression.
sound	Function	165	Returns the name of the sound that is currently playing.
space	Constant	213	The space character (ASCII 32); same as the literal " ".
spray [can]	Tool	94, 170	Name of tool from Tools palette; used with choose command or returned by the tool function.
sqrt	Function	166	Returns the square root of a number.
stack	Object	38	Generic name of stack object; used with specific name (go to stack "help").
startUp	System message	80	Sent to the current card (first card of the Home stack) when HyperCard first begins running.
style	Property	199	Determines the style of a specified field or button. (See also page 207.)
subtract	Command	131	Subtracts the value of an expression from the value in a container.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
suspend	System message	83	Sent to the current card when HyperCard is suspended by launching another application with the open command.
tab	Constant	213	The horizontal tab character (ASCII 9).
tabKey	Command	131	Places the insertion point in the next unlocked field on the current background and card.
tabKey	System message	81	Sent to current card when Tab key is pressed.
tan	Function	166	Returns the tangent of an angle.
target	Function	167	Indicates the object that initially received the message that initiated execution of the current handler.
ten	Constant	213	String representation of the numerical value 10.
tenth	Ordinal	36	Designates object or chunk number ten within its enclosing set.
text	Tool	94, 170	Name of tool from Tools palette; used with choose command or returned by the tool function.
textAlign	Property	188	Determines the alignment of characters created with the Paint Text tool, or those in a field, or those in the name of a button. (See also pages 200 and 208.)
textArrows	Property	180	Determines the functions of the arrow keys.
textFont	Property	188	Determines the font of characters created with the Paint Text tool, or those in a field, or those in the name of a button. (See also pages 200 and 208.)
textHeight	Property	188	Determines the space between the baseline and characters created with the Paint Text tool or those in a field. (See also page 201.)
textSize	Property	189	Determines the size of Paint text or text in a field or in the name of a button. (See also pages 201 and 209.)
textStyle	Property	189	Determines the style of Paint text or the text in a field or in the name of a button. (See also pages 202 and 209.)

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
`the`	Special	138	Precedes a function name to indicate a function call to one of HyperCard's built-in functions. You can't call a user-defined function with `the`. Also allowed, but not required, preceding special container names (`the Message box`) and properties.
`then`	Keyword	70	Follows the conditional expression in an `if` structure to introduce the action clause.
`third`	Ordinal	36	Designates object or chunk number three within its enclosing set.
`this`	Object modifier	37	Used with `card`, `background`, or `stack` to refer to the current one.
`three`	Constant	213	String representation of the numerical value 3.
`ticks`	Function	168	Determines the number of ticks since the Macintosh was turned on or restarted.
`time`	Function	169	Returns the current time as a text string.
`tool`	Function	170	Returns the name of the currently chosen tool.
`true`	Constant	213	Boolean value resulting from evaluation of a comparative expression and returned from some functions.
`trunc`	Function	171	Determines the integer part of a number.
`two`	Constant	213	String representation of the numerical value 2.
`type`	Command	132	Inserts the specified text at the insertion point.
`up`	Constant	213	Value returned by various functions to describe the state of a key or the mouse button.
`userLevel`	Property	181	Determines the user level from 1 to 5.
`value`	Function	172	Evaluates an expression.
`version`	Function	172	Returns the version number of the currently running HyperCard application.
`visible`	Property	183	Determines whether or not a window, field, or button appears on the screen. (See also pages 202 and 209.)
`visual`	Command	133	Sets up a specified visual transition to the next card opened.
`wait`	Command	135	Causes HyperCard to pause before executing the rest of the current handler.

Table H-1 *(continued)*
HyperTalk vocabulary

Term	Category	Page	Meaning
`wideMargins`	Property	202	Determines whether or not additional space is displayed in the margins of a specified field.
`word`	Chunk	54	Piece of text in delimited by spaces in any container or expression.
`words`	Chunk type	154	Specifies words as type of chunk to the `number` function.
`write`	Command	136	Copies specified text into a specified disk file.
`zero`	Constant	213	String representation of the numerical value 0.

Glossary

actual parameters: See **parameters.**

background: A type of HyperCard object; a basic template which is shared by a number of cards. The background is composed of the **background picture, background field,** and **background button.**

background button: A button that belongs to a background; it appears on, and its actions are the same for, all cards with the same background. Contrast with **card button.**

background field: A field that belongs to a background; its size, position, and text attributes remain constant on all cards associated with that particular background, but its text changes from card to card. Contrast with **card field.**

background picture: A picture that belongs to a background; it applies to a series of cards. You see the Background picture by choosing Background from the Edit menu. Contrast with **card picture.**

browse: To wander through HyperCard's stacks.

Browse tool: The tool you use to click buttons and to set the insertion point in fields.

button: A type of HyperCard object; an action object or "hot spot" on the screen. For example, clicking a button with the Browse tool can take you to the next card. See also **background button, card button.**

Button tool: The tool you use to create, change, and select buttons.

card: A type of HyperCard object; HyperCard's basic unit of information.

card button: A button that belongs to a card; it appears on, and its actions apply to, a single card. Contrast with **background button.**

card field: A field that belongs to a card; its size, position, text attributes, and contents are limited to the card on which the field is created. Contrast with **background field.**

card picture: A picture that belongs to and which applies only to a specific card. Contrast with **background picture.**

chunk: A piece of the character string representing a value. Valid chunks are characters, words, items, and lines.

Command key: The key at the lower-left side of the keyboard that has a propeller-shaped symbol. On some keyboards this key also has an Apple symbol and might be called the *Apple key*.

command: A response to a particular message; a command is a built-in message handler residing in HyperCard. See also **external command.**

constant: A named value that never changes. For example, the constant `empty` stands for the null string, a value that can also be represented by the literal expression `""`.

container: A place where you can store a value. Containers are: **fields,** the **Message box,** the **selection,** and **variables.**

control structure: A block of HyperTalk statements defined with keywords that enables you to control the order or the conditions under which it executes.

current: (adj.) The card, background, or stack you're using now. For example, the current card is the one you can see on your screen.

dynamic path: A series of extra objects inserted into the path through which a message passes when its **static path** does not include the current card. The dynamic path comprises the current card, current background, and current stack.

expression: A description of how to get a value; a **source of value** or complex expression built from sources and operators.

external command: A command written by a programmer to extend HyperCard's built-in command set, attached to a stack or in HyperCard.

factor: A single element of value in an expression. See also **value.**

field: A **container** in which you type regular (as opposed to Paint) text. Also, the tool you use to create a field. HyperCard has two kinds of fields—**card fields** and **background fields.**

Field tool: The tool you use to create, change, and select fields.

formal parameters: See **parameter variables.**

function: A named value that HyperCard calculates each time it is used. The way in which the value is calculated is defined internally for HyperTalk's built-in functions, and you can define your own functions with function handlers.

function call: The use of a function name in a HyperTalk statement or in the Message box, invoking either a function handler or a built-in function.

function handler: A handler that executes in response to a function call matching its name.

General tool: Any HyperCard tool that isn't a Paint tool. The General tools are Browse, Button, and Field.

global properties: The properties that determine aspects of the overall HyperCard environment. For example, `userLevel` is a global property which determines the current **user level** setting.

global variable: A variable that is valid for all handlers in which it is declared with the `global` keyword. Contrast with **local variable.**

handler: A block of HyperTalk statements contained in the script of an object that executes in response to a message or a function call matching the handler's name. HyperTalk has **message handlers** and **function handlers.**

hierarchy: See **object hierarchy.**

Home card: The first card in the Home stack; it is generally used as a pictorial index to stacks. Choose Home from the Go menu to get to Home (or press Command-H). You can also type `go home` in the Message box or include it as a statement in a handler.

HyperTalk: HyperCard's built-in script language for HyperCard users.

identifier: A character string of any length, beginning with an alphabetic character, containing any alphanumeric character and, optionally, the underscore character. Identifiers are used for **variable** and **handler** names.

keyboard equivalent key: A key you press together with the Command key to issue a menu command.

keyword: Any one of the 14 words that have a predefined meaning in HyperTalk. Examples of keywords are `on`, `if`, `do`, and `repeat`.

layer: The order of a button or field relative to other buttons or fields on the same card or background. The object created most recently is ordinarily the topmost object (that is on the front layer).

literal: An expression denoted by double quotation marks at either end of a character string; its value is the string itself.

local variable: A variable that is valid only within the handler in which it is used (local variables need not be declared). Contrast with **global variable.**

message: A character string you send to an object from a **script** or the Message box, or which HyperCard sends in response to an event. Some examples of HyperTalk messages are `mouseUp`, `go`, and `push card`.

Message box: a **container** that you use to send messages to objects or to evaluate expressions.

message handler: A handler that executes in response to a message matching its name.

number: a character string consisting of any combination of the numerals 0 through 9, optionally including one period (.) representing a decimal value. A number can be preceded by a hyphen or a minus sign to represent a negative value.

object: An element of the HyperCard environment that sends and receives messages. There are five kinds of HyperCard objects: **buttons, fields, cards, backgrounds,** and **stacks.**

object descriptor: Designation used to refer to an object. An object descriptor is formed by combining the name of the type of object with a specific name, number, or ID number. For example, `background button 3` is an object descriptor.

object hierarchy: The ordering of HyperCard objects that determines the path through which **messages** pass.

object properties: The properties that determine how HyperCard objects look and act. For example, the `location` property of a button determines where it appears on the screen.

on-line help: assistance you can get from an application program while it's running. In this guide, on-line help refers to HyperCard's disk-based Help system.

operator: a HyperTalk language element that you use in an **expression** to manipulate or calculate **values.**

Paint text: Text you type using the Paint Text tool. Paint text can appear anywhere, while **regular text** must appear in a field created with the Field tool. When you finalize Paint text by clicking, it becomes part of a card or background picture.

Paint tool: Any HyperCard tool you use to make pictures. Tools include Lasso, Brush, Spray, Eraser, and many others.

painting properties: The properties that control aspects of HyperCard's painting environment, which is invoked when you choose a Paint tool. For example, the `brush` property determines the shape of the Brush tool.

palette: The name for a **tear-off menu** when it's been torn off. A palette remains visible on the screen so you can use it without having to pull down the menu. HyperCard has two palettes—Tools and Patterns.

parameters: Values passed to a handler by a message or function call. Any expressions after the first word in a message are evaluated to yield the parameters; the parameters to a function call are enclosed in parentheses or, if there is only one, it can follow `of`.

parameter variables: Local variables in a handler which receive the values of parameters passed with the message or function call initiating the handler's execution.

picture: Any graphic or part of a graphic, created with a Paint tool or imported from an external file, which is part of a card or background.

point: In printing, the unit of measurement of the height of a text character; one point is about 1/72 of an inch. When you select a font, you can also select a point size, such as 10-point, 12-point, and so on. Also, a location on the screen described by two integers, separated by a comma, representing horizontal and vertical offsets, measured in pixels from the top-left corner of the card window or (in the case of the card window itself) of the screen.

power key: One of a number of keys on the Macintosh keyboard you can press to initiate a menu action when a Paint tool is active. Power keys are enabled when you choose Power Keys from the Options menu or you check Power Keys in the User Preferences card in the Home stack.

properties: The defining characteristics of any HyperCard object and of HyperCard's environment. See also **global properties, object properties, painting properties,** and **window properties.**

Recent: A special dialog box that holds pictorial representations of the last 42 unique cards viewed. Choose Recent from the Go menu to get the dialog box. Also, as in `recent card`, an adjective describing the card you were on immediately prior to the current card.

recursion: The continued repeating of an operation or group of operations. Recursion occurs when a handler calls itself.

regular text: Text you type in a field. You use the Browse tool to set an insertion point in a field and then type. Regular text is editable and searchable, while **Paint text** is not.

script: A collection of handlers written in HyperTalk and associated with a particular object.

search path: The route the computer must follow to retrieve a file you ask for.

selection: A container that holds the currently selected area of text. Note that text found by the `find` command is not selected. See also **container.**

source of value: HyperTalk's most basic expressions; the language elements from which values can be derived: **constants, containers, functions, literals,** and **properties.**

stack: A type of HyperCard object which is a collection of cards; a HyperCard document. See also **card.**

static path: The message-passing route defined by an object's own hierarchy. For example, the static path followed by a message sent to (but not handled by) a button would include the card to which the button belongs, the background associated with that card, and the stack containing them. Contrast with **dynamic path.**

System file: Software your computer uses to perform its basic operations.

system message: Message sent to an object by HyperCard in response to an event such as a mouse click or the creation or deletion of an object.

target: The object which first receives a message.

tear-off menu: A menu that you can convert to a **palette** by dragging the pointer beyond the menu's edge. HyperCard has two tear-off menus—Tools and Patterns.

text field: See **field.**

text property: A quality or attribute of a character's appearance. Properties include style, font, and size.

tool: An implement you use to do work. HyperCard has tools for browsing through cards and stacks, creating text fields, editing text, making buttons, and creating and editing pictures.

user level: The property of HyperCard ranging from 1 to 5, usually chosen on the User Preferences card in the Home stack, that lets you use HyperCard's tools and abilities. The five user levels are: Browsing, Typing, Painting, Authoring, and Scripting.

value: The information on which HyperCard operates. All HyperCard values can be treated as strings as characters.

variable: A named container that can hold a value consisting of a character string of any length. HyperCard has **local variables** and **global variables.** See also **container.**

window properties: The properties that determine how the Message box and the Tool and Pattern palettes are displayed. For example, the `visible` property determines whether or not the specified window is displayed on the screen.

Index

Note: See Appendix H, "HyperTalk Vocabulary," for brief descriptions of HyperTalk keywords, commands, and other terms.

Cast of Characters

A

B

C

G

H

I, J

K

T

U

V

W

X, Y

Z

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word. Proof pages were created on the Apple LaserWriter® Plus. Final pages were created on the Varityper® VT600™. POSTSCRIPT®, the LaserWriter page-description language, was developed by Adobe Systems Incorporated. Some of the illustrations were created using Adobe Illustrator™.

Text type is ITC Garamond® (a downloadable font distributed by Adobe Systems). Display type is ITC Avant Garde Gothic®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.

ISBN 0-201-17632-7