Inside Special K

Written by Robert Berkowitz
N&C Publications
Beta Draft
June 22, 1989
Apple Confidential

APPLE COMPUTER, INC.

© Apple Computer, Inc., 1988, 1989
20525 Mariani Avenue
Cupertino, CA 95014
(408) 996-1010

Contents

Figures and Tables

Figure 7-8 Network look-up dialog box / 178

Preface

About this document

Inside Special K provides definitive information for application software developers, communications tools developers, and hardware developers who want to use services provided by Special K. For application software developers, this document describes and shows how to use the four Special K managers that make it easier to write communications software for the Apple® Macintosh®. For communications tools developers, this document shows how to develop communications tools that can be used by the Special K managers. And for hardware developers, this document shows what protocols to follow to register hardware—like internal modems or serial cards—with the Special K Communications Resource Manager.

This document contains an overview of Special K in Chapter 1 and a sample application that uses Special K in Chapter 2. The next five chapters discuss the Special K managers and utilities. This is where the routines and data structures that an application uses are described. Chapter 8 through Chapter 11 tells you how to create a tool to add to Special K. While tool developers will be interested in reading these chapters, application developers may have little need to read them. Appendix A contains guidelines that communications tool developers should read to insure that the tools they create are fully compatible with Special K. Appendix B shows you sample code solutions to common programming problems.

Inside Special K is written for experienced programmers. It is assumed that you know both how to program the Macintosh and have some familiarity with communications or networking applications. To use each manager requires specific programming knowledge; suggestions on where to find more infomation are included at the beginning of each chapter. In addition, the following section suggests reference information that will help you understand the technical concepts used in this document.

Where to Go for More Information

Refer to the following Addison-Wesley books for additional information about the subjects covered in this manual:

- *Designing Cards and Drivers for Macintosh II and Macintosh SE*
- *Human Interfaces Guidelines: The Apple Desktop Interface*
- *Inside Macintosh* (Volumes I-V, X-Ref)
- *Programmer's Introduction to Macintosh*

- *Technical Introduction to Macintosh*
- *Inside AppleTalk*

You may also refer to the following documents from APDA™:

- *Software Development for International Markets: A Technical Reference*
- *Macintosh Technical Notes*

APDA provides a wide range of development products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. For information about APDA, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33-G
Cupertino, CA 95014-6299

(800) 282-APDA (800-282-2732)
Fax: 408-562-3971
Telex: 171-576
AppleLink: APDA

If you plan to develop Apple-compatible hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51-W
Cupertino, CA 95014-6299

Conventions

The following notations are used in this document to draw attention to particular items of information:

◆	*Note:*	a note that may be interesting or useful
◆	*Assembly Note:*	a note of interest to assembly-language programmers only
△	**Important**	a note that is particularly important
▲	**Warning**	a point that you need to be cautious about

Names of routines (procedures or functions), constants, or code fragments appear in `courier` typeface.

Chapter 1 **About Special K**

About this Chapter

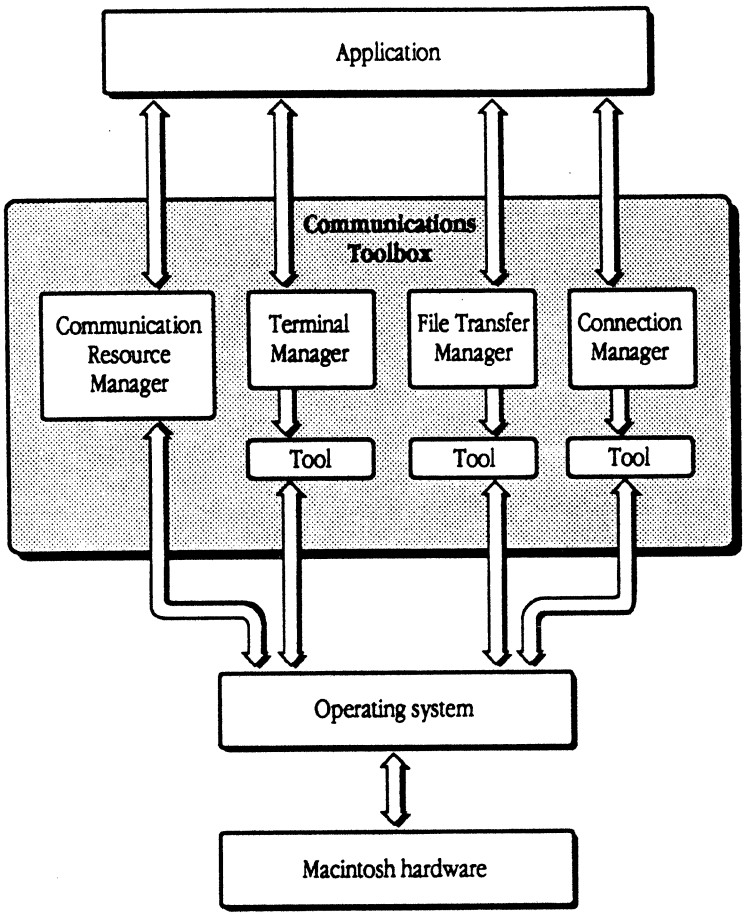
This chapter gives you an overview of Special K. It starts out by telling you about the managers and utilities that are part of Special K, and then discusses a fundamental concept, the difference between routines and tools. The last part of the chapter provides technical details, such as system hardware and software requirements, as well as how to install Special K.

What Comprises Special K

Inside of Special K are four managers and one set of utilities . These managers and utilities are an extension to the Macintosh Toolbox and provide basic networking and communications services. Just as the Macintosh Toolbox makes it easier for you to develop stand-alone Macintosh applications, Special K helps you develop networking and communications applications.

Each of the managers in Special K handles a different aspect of networking and communications: connection management, terminal emulation management, file transfer management, and communications resource management. The managers provide routines that your application can call to indirectly interact with the operating system. Figure 1-1 shows how Special K fits between your application and the operating system.

Figure 1-1 Where Special K fits in



While the managers in Special K handle distinctly different aspects of networking and communication, your application might need to call routines from more than one of the managers to implement a feature. For instance, in order to perform terminal emulation, your program might make use of Connection Manager routines to maintain the data connection and Terminal Manager routines to handle the specifics of the terminal emulation. Using routines from more than one of the managers in your application is an acceptable development technique.

However, your application does not have to use Special K routines to perform all of its networking and communications functionality; your application can maintain the data connection itself and use only the Terminal Manager to perform a terminal emulation. Keep in mind, though, that by not using Special K routines, your application will not be as modular and might have a more difficult time interfacing with new tools as they become available.

Routines and tools: What's the difference?

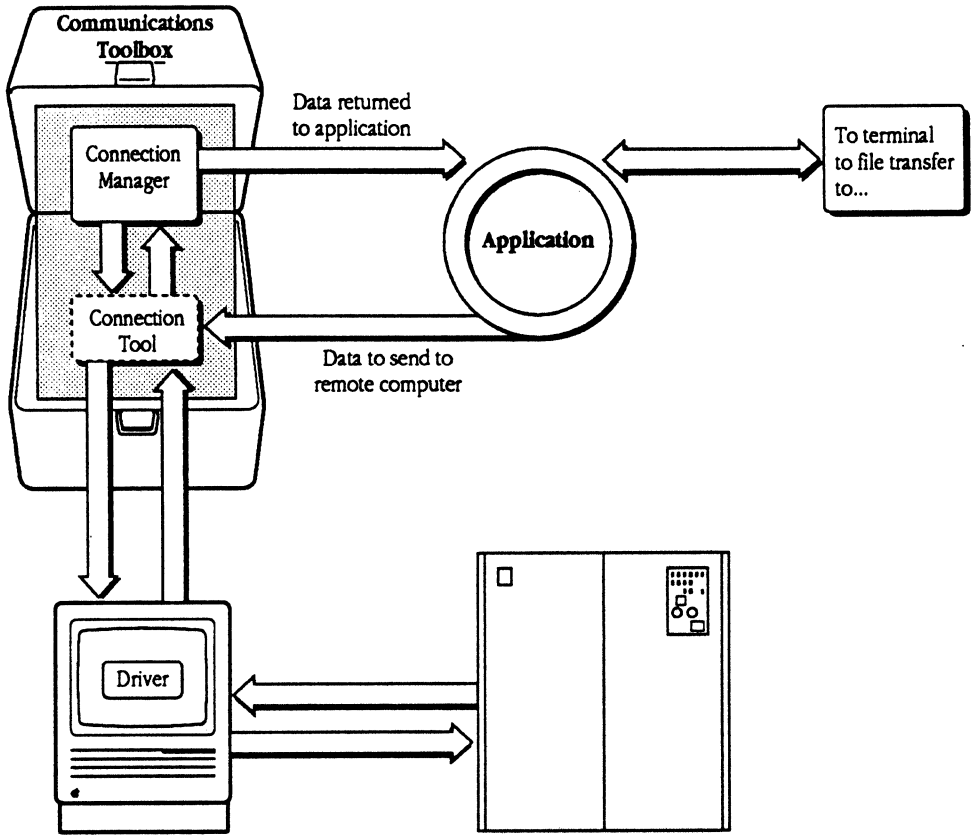
There are two interfaces (besides the user interface) to consider when programming with Special K. One is the interface between the application and Special K and the other is between Special K and the Macintosh operating system.

The interface between an application and Special K is defined by the routines in each of the managers. By calling routines, an application can request basic networking and communications services. If you are writing applications (not tools), this is the interface with which you need to be most concerned; it is discussed in Chapter 3 through Chapter 7.

The interface between Special K and the Macintosh operating system is controlled by *tools*. Tools are units of code that implement requested networking and communications services. When an application calls a Special K routine, it does so without concern for the underlying protocols. It is the job of the tool to implement basic networking and communications services according to a specific protocol. If you are writing tools (not applications), this is the interface with which you need to be most concerned; it is discussed in Chapter 8 through Chapter 11. Tool writers need to read at least two of these chapters: Chapter 8, which discusses concepts common to all types of tools, and one of the other chapters that deals with a specific type of tool.

Figure 1-2 shows the interaction between an application and one of the Special K managers, which in this case is the Connection Manager. Notice that the application interfaces with the Connection Manager, which in turn interfaces with the connection tool. The connection tool, in turn, communications with a driver and passes back to the application any relevant information (a complete discussion of the Connection Manager is in Chapter 3, "Connection Manager").

Figure 1-2 How Special K managers interact with applications and tools.



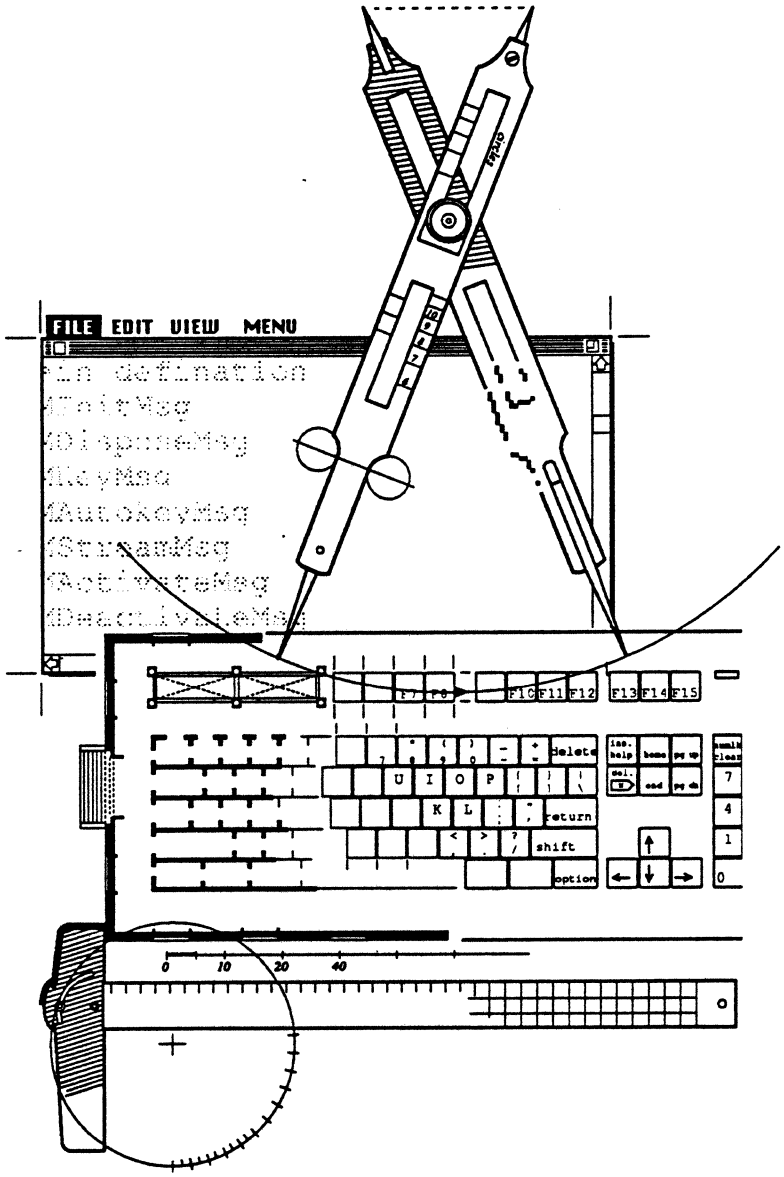
Technical Details

Special K can be run on all Macintosh computers that have at least 1 Mb of RAM, the Macintosh Plus (128K) ROMs or later, and System 6.0.2 or later. Minimum disk space requirements are two floppy disks, a single FDHD floppy disk, or a hard disk (which is recommended).

- *Note:* On a 1Mb Macintosh running MultiFinder™ , you may only be able to run a single application with the Finder coresident. You also may be unable to run HyperCard® under MultiFinder on a 1Mb Macintosh with Special K installed.

You can install additional tools into Special K by dragging the icon for the tool into the Communications folder, which is inside the System folder.

Chapter 2 **Programming with Special K**



About this chapter

This chapter provides an example of how an application can use Special K to implement communications services.

The sample code is not a complete program. Instead, it contains the parts of a program that handle communication functions; the rest of the program has been replaced with comments to limit the focus to programming Special K rather than Macintosh programming in general. This sample shows you where in an application to put the hooks to which you can attach Special K routines.

The sample application (if it were a real, working program) allows you to perform functions that span the three major Special K managers : the Connection Manager, the File Transfer Manager, and the Terminal Manager. Specifically, the application allows you to:

- Open and close a connection,
- Send and receive files,
- Configure connections, terminal emulations, and file transfers
- Clear the screen
- Reset the terminal.

The sample code is split into three sections to make it easier to understand. The first section shows how your application can deal with events that result from menu selections; the sample application contains routines that handle basic communications services, like opening a connection and sending a file. The second section shows how your application can deal with events like scrolling and mouse clicks. The last section shows the sample application's main code loop.

Code that handles menu events

Handling menu choices

```
PROCEDURE DoCommand(mResult : LONGINT);
VAR
      theItem             : INTEGER;
      theMenu             : INTEGER;
      theName             : STR255;
      savedPort           : GrafPtr;
      garbage             : INTEGER;
      err                       : OSErr;

BEGIN
```

```
        theItem := LoWord(mResult);
        theMenu := HiWord(mResult);

        theTerm := GethTerm(FrontWindow);        { is it a terminal? }
        IF theTerm <> NIL THEN
                IF TMMenu(theTerm, theMenu, theItem) THEN
                        Exit(DoCommand);                        { yup }

        theConn := GethConn(FrontWindow);        { is it a connection? }
        IF theConn <> NIL THEN
                IF CMMenu(theConn, theMenu, theItem) THEN
                        Exit(DoCommand);                        { yup }

        theFT := GethFT(FrontWindow);     { is it a file transfer? }
        IF theFT <> NIL THEN
                IF FTMenu(theFT, theMenu, theItem) THEN
                        Exit(DoCommand);                        { yup }
        { menus are mine here! }
END;
```

Opening a connection

```
PROCEDURE DoInitiate;
VAR
        theWindow : WindowPtr;
        theErr : CMErr;

        sizes : BufferSizes;
        status : LONGINT;

BEGIN
{DebugStr('enterring DoInitiate');}
        dirtyMenu := TRUE;
        theWindow := FrontWindow;
        theConn := GethConn(theWindow);
        IF theConn = NIL THEN
                BEGIN
                SysBeep(5);
                Exit(DoInitiate);
                END;

        theErr := CMStatus(theConn, sizes, status);

        IF BAND(status, CMStatusOpen + CMStatusOpening) = 0 THEN
                theErr := CMOpen(theConn, FALSE, NIL, 0);
END;
```

Closing the connection

```
PROCEDURE DoClose;
VAR
	theWindow : WindowPtr;
	theWIndowPeek : WindowPeek;
	pWindow : WindowP;

BEGIN
	theWindow := FrontWindow;

	IF theWindow <> NIL THEN
	BEGIN
		theWindowPeek := WindowPeek(theWindow);
		garbage := theWindowPeek^.windowKind;
		IF garbage < 0 THEN
			CloseDeskAcc(garbage)
		ELSE
		BEGIN
			pWindow := WindowP(WindowPeek(theWindow)^.refCon);

			theTerm := GethTerm(theWindow);
			IF theTerm <> NIl THEN
			BEGIN
				TMDispose(theTerm);
			END;
			theConn := GethConn(theWindow);
			IF theConn <> NIl THEN
			BEGIN
				CMDispose(theConn);
			END;

			theFT := GethFT(theWindow);
			IF theFT <> NIL THEN
			BEGIN
				FTDispose(theFT);
			END;

			DisposPtr(Ptr(pWindow));
			DisposeWindow(theWindow);

		END;
	END;
END;
```

Killing the connection

```
PROCEDURE DoKill;
VAR
        theWindow : WindowPtr;
        theErr : CMErr;
        sizes : BufferSizes;
        status : LONGINT;

BEGIN
        dirtyMenu := TRUE;
        theWindow := FrontWindow;
        theConn := GethConn(theWindow);
        IF theConn = NIL THEN
                BEGIN
                SysBeep(5);
                Exit(DoKill);
                END;

        theErr := CMStatus(theConn, sizes, status);

        IF BAND(status, CMStatusOpen + CMStatusOpening) <> 0 THEN
                BEGIN
                theErr := CMClose(theConn, FALSE, NIL, 0, TRUE);
                IF theErr <> 0 THEN
                        SysBeep(5);
                        {
                        DebugStr('there was a problem closing');
                        }
                END
        ELSE
                BEGIN
                SysBeep(5);SysBeep(5);
                {
                DebugStr('cannot close for some reason');
                }
                END;
END;
```

Sending files

```
PROCEDURE DoSend;
VAR
        theWindow       : WindowPtr;
        theReply        : SFReply;
        where           : Point;
        numTypes        : INTEGER;
```

```
        typeList      : SFTypeList;
        pWindow             : windowP;
        anyErr              : OSErr;

BEGIN
        theWindow := FrontWindow;
        pWindow := windowP(GetWRefCon(theWindow));
        theFT := GethFT(theWindow);
        IF theFT = NIL THEN
                BEGIN
                SysBeep(5);
                Exit(doSend);
                END
        ELSE BEGIN
                SetPt(where, 100, 100);

                typeList[0] := 'TEXT';
                numTypes := -1;
                IF BAND(theFT^^.attributes, FTTextOnly) <> 0 THEN
                        numTypes := 1;

                SFGetFile(where, 'A Cruel Moose, indeed', NIL, numTypes,
                typeList, NIL, theReply);
                if theReply.good then begin
                        anyErr := FTStart(theFT,FTTransmitting,theReply);
                        if (anyErr <> noErr) then
                                SysBeep(5);
                end;
        END;
END;
```

Receiving Files

```
PROCEDURE DoReceive;
VAR
        theWindow : WindowPtr;
        theReply : SFReply;
        where : Point;
        typeList : SFTypeList;
        anyErr        : OSErr;
        pWindow             :             WindowP;
        theConn:            ConnHandle;

BEGIN
        theWindow := FrontWindow;
        theFT := GethFT(theWindow);
        pWindow := WindowP(WindowPeek(theWindow)^.refCon);
```

```
        IF theFT = NIL THEN BEGIN
                SysBeep(5);
                Exit(DoReceive);
        END
        ELSE BEGIN
                theReply.vRefNum := 0;
                theReply.fName := '';

                theConn := GethConn(theWindow);
                IF theConn <> NIL THEN
                        IF theFT^^.autoRec <> '' THEN
                        BEGIN
                                CMRemoveSearch(theConn, pWindow^.searchBlk);
                                pWindow^.searchBlk := 0;
                        END;

                anyErr := FTStart(theFT,FTReceiving,theReply);
                if (anyErr <> noErr) then
                        SysBeep(5);
        END;
END;
```

Configuring a connection

```
PROCEDURE DoConnectionConfig;
VAR
        theWindow       :       WindowPtr;
        theConn                 :       ConnHandle;
        result                  :       INTEGER;
        where           :       Point;

BEGIN
        SetPt(where, 10, 40);
        theWindow := FrontWindow;

        theConn := GethConn(theWindow);
        IF theConn = NIL THEN
                BEGIN
                {DebugStr('No Connection Record');}
                Exit(DoConnectionConfig);
                END;
        result := CMChoose(theConn, where, NIL);
        SethConn(theWindow, theConn);
        IF result = ChooseDisaster THEN
        BEGIN
                DoDeepshit;
                DoClose;
```

```
        END;
        IF (result = ChooseOKMajor) OR (result = ChooseOKMinor) THEN
                DirtyDocument(theWindow);
END;
```

Configuring a terminal emulation

```
PROCEDURE DoTerminalConfig;
VAR
        theWindow : WindowPtr;
        theTerm             :       TermHandle;
        result              :       INTEGER;
        where       :       Point;

BEGIN
        theWindow := FrontWindow;

        SetPt(where, 10, 40);

        theTerm := GethTerm(theWindow);
        IF theTerm = NIL THEN
                BEGIN
                {DebugStr('No Terminal Record');}
                Exit(DoTerminalConfig);
                END;
        result := TMChoose(theTerm, where, NIL);
        SethTerm(theWindow, theTerm);
        IF result = ChooseDisaster THEN
        BEGIN
                DoDeepshit;
                DoClose;
        END;

        IF (result = ChooseOKMajor) OR (result = ChooseOKMinor) THEN
                DirtyDocument(theWindow);
END;
```

Configuring a file transfer

```
PROCEDURE DoFileTransferConfig;
VAR
        theWindow     :             WindowPtr;
        theFT         :             FTHandle;
        theConn             :             ConnHandle;
        result              :             INTEGER;
        where         :             Point;
```

```
        pWindow             :           WindowP;
        tempString    :           STR255;

BEGIN
        SetPt(where, 10, 40);
        theWindow := FrontWindow;
        pWindow := WindowP(WindowPeek(theWindow)^.refCon);

        theFT := GethFT(theWindow);
        theConn := GethConn(theWindow);
        IF theFT = NIL THEN
                BEGIN
                {DebugStr('No File Transfer Record');}
                Exit(DoFileTransferConfig);
                END;
        result := FTChoose(theFT, where, NIL);
        SethFT(theWindow, theFT);
        IF result = ChooseDisaster THEN
        BEGIN
                DoDeepshit;
                DoClose;
                Exit(DoFileTransferConfig);
        END;

        IF (result = ChooseOKMajor) OR (result = ChooseOKMinor) THEN
                DirtyDocument(theWindow);

        tempString := theFT^^.autoRec;
        if (tempString <> '') then
        begin
        { store the searchblock some where, so we can remove it later }
                IF BAND(theFT^^.flags, FTIsFTMode) =  0 THEN
                        IF pWindow^.searchBlk = 0 THEN
                                pWindow^.SearchBlk := CMAddSearch(theConn,
                                tempString, 0, @AutoRecCallback);
                        IF pWindow^.searchBlk = -1 THEN
                        BEGIN
                                DebugStr('cannot add search');
                                pWindow^.searchBlk := 0;
                        END;
        end
        else
        begin
                {  grap searchblock from the place stored it }
                { assume if the search block was not added before,
                CMRemoveSearch do nothing -  check this later }
                IF pWindow^.searchBlk <> 0 THEN
                        CMRemoveSearch(theConn,pWindow^.SearchBlk);
```

```
            pWindow^.searchBlk  :=  0;
        end;

END;
```

Making a new session document

```
PROCEDURE  MakeNew(termID,ftID,connID:  integer);
VAR
        err                          : OSErr;

        theWindow             : WindowPtr;
        theWIndowPeek         : WindowPeek;
        pWindow               : WindowP;
        theRect               : Rect;
        sizes                 : BufferSizes;
        termEnvironment       : TermEnvironRec;
        testV:  LONGINT;
        boof:  STR255;

BEGIN
        theWindow  :=  GetNewWindow(128,  NIL,  POINTER(-1));

        SetPort(theWindow);
        ClipContent(theWindow);
        TextFont(monaco);

        pWindow  :=  WindowP(NewPtr(SIZEOF(WindowR)));

        theRect  :=  theWindow^.portRect;
        InsetRect(theRect,  3,  3);
        theRect.right  :=  theRect.right  -  15;
        theRect.bottom  :=  theRect.bottom  -  15;

        SetWRefCon(theWindow,  LONGINT(pWindow));

        WITH  pWindow^  DO
        BEGIN
                hTerm  :=  NIL;
                hConn  :=  NIL;
                hFT  :=  NIL;
        END;

        if  (termID  =  GET_FIRST_TOOL)  then     { Get the first tool     }
                termID  :=  FindToolID(NEWTM);

        if  (termID  =  GET_FIRST_TOOL)  then        { No tools found }
                Exit(MakeNew);
```

```
pWindow^.hTerm := TMNew(theRect, theRect, TMSaveBeforeClear,
      termID, theWindow, @sendProc, @cacheProc, @breakProc, NIL,
      @TermGetConnEnvirons, 0, 0);

if (pWindow^.hTerm = nil) then
BEGIN
      DebugStr('no TMNew found');
      Exit(MakeNew);
END;

pWindow^.cachelinecount := 0 ;
pWindow^.viscacheline := 0 ;
pWindow^.cachetopline := 0 ;
pWindow^.clineInfoPtr := mylineInfoPtr( 0 );

theRect := theWindow^.portRect;

sizes[CMDataIn] := 1024;
sizes[CMDataOut] := 1024;
sizes[CMCntlIn] := 0;
sizes[CMCntlOut] := 0;
sizes[CMAttnIn] := 0;
sizes[CMAttnOut] := 0;

if (connID = GET_FIRST_TOOL) then       { Get first tool }
      connID := FindToolID(NEWCM);

if (connID = GET_FIRST_TOOL) then             {No tools found }
BEGIN
      DebugStr('no CMNew found');
      Exit(MakeNew);
END;

pWindow^.hConn := CMNew(connID, CMData, sizes, 0, 0);
if (pWindow^.hConn = nil) then
      Exit(MakeNew);
CMSetRefCon(pWindow^.hConn, LONGINT(theWindow));

TMSetRefCon(pWindow^.hTerm, LONGINT(theWindow));

if (ftID = GET_FIRST_TOOL) then               { Get first tool}
      ftID := FindToolID(NEWFT);

if (ftID = GET_FIRST_TOOL) then               { No tools found}
BEGIN
      DebugStr('no file transfer found');
      Exit(MakeNew);
```

```
        END;

        pWindow^.hFT := FTNew(ftID, 0, @FTsendProc, @FTreceiveProc, NIL,
        NIL, @FTGetConnEnvirons,theWindow, ORD4(theWindow), 0);
        if (pWindow^.hFT = nil) then
                Exit(MakeNew);

        pWindow^.searchBlk := 0;
        if (pWindow^.hFT^^.AutoRec <> '') then
        begin
        { store the searchblock some where, so we can remove it later }
                IF BAND(pWindow^.hFT^^.flags, FTIsFTMode) =  0 THEN
                BEGIN
                        boof := pWindow^.hFT^^.autoRec;
                        pWindow^.SearchBlk := CMAddSearch(pWindow^.hConn,
                        boof, 0, @AutoRecCallback);
                        IF pWindow^.searchBlk = -1 THEN
                        BEGIN
                                DebugStr('cannot add search');
                                pWindow^.searchBlk := 0;
                        END;
                END;
        end;

        FTSetRefCon(pWindow^.hFT, LONGINT(theWindow));

        pWindow^.wasFTMode := FALSE;
        pWindow^.startFT := FALSE;
END;
```

Code that handles other events

Activate events

```
PROCEDURE DoActivate(theEvent : EventRecord);
VAR
        theWindow       : WindowPtr;
        processed       : BOOLEAN;
        hScroll,
        vScroll                 : ControlHandle;
        savedPort:      GrafPtr;

BEGIN
        GetPort(savedPort);
```

```
        theWindow := WindowPtr(theEvent.message);
        SetPort(theWIndow);

        processed := BAnd(theEvent.modifiers, activeFlag) <> 0;
        theTerm := GethTerm(theWindow);
        theConn := GethConn(theWIndow);
        theFT := GethFT(theWIndow);

        hScroll := GetHScroll(theWindow);
        vScroll := GetVScroll(theWindow);
        IF theTerm <> NIL THEN
                TMActivate(theTerm, processed);

        IF theConn <> NIL THEN
                CMActivate(theConn, processed);
        IF theFT <> NIL THEN
                FTActivate(theFT, processed);
END;
```

Resume events

```
PROCEDURE DoResume(theEvent : EventRecord);
CONST
        resumeFlag = 1;
        scrapModifiedFlag = 2;

VAR
        theWindow : WindowPtr;
        boo : BOOLEAN;                                  { resuming }
        savedPort       :       GrafPtr;

BEGIN
        theWindow := FrontWindow;
        IF theWindow <> NIL THEN
                BEGIN
                boo := BAND(theEvent.message, resumeFlag) <> 0;
                IF boo THEN
                BEGIN
                        SetPort(theWindow);
                        IF scrapVis THEN
                        BEGIN
                                ShowWindow(scrapWindow);
                                GetPort(savedPort);
                                SetPort(scrapWindow);
                                InvalRect(scrapWindow^.portRect);
                                SetPort(scrapWindow);
                        END
```

```
            END
            ELSE
            BEGIN
                  IF scrapVis THEN
                        HideWindow(scrapWindow);
            END;

            theTerm := GethTerm(theWindow);
            IF theTerm <> NIL THEN
                  TMResume(theTerm, boo);
            theConn := GethConn(theWindow);
            IF theConn <> NIL THEN
                  CMResume(theConn, boo);
            theFT := GethFT(theWindow);
            IF theFT <> NIL THEN
                  FTResume(theFT, boo);

            END; {if window nil }
END;
```

Update events

```
PROCEDURE DoUpdate(theEvent : EventRecord);
VAR
      theWindow     : WindowPtr;
      savedPort     : GrafPtr;
      savedClip     : RgnHandle;
      pWindow             : windowP;
      termEnv             : termEnvironRec;    { terminal environment }
      err                 : OSErr ;
      drawRect      : Rect ;
      oldptr1             : Ptr ;
      oldptr2             : Ptr ;
      i                   : INTEGER ;
      skip          : INTEGER ;
      curwidth      : INTEGER ;
      oldlinesize   : INTEGER ;
      alineInfoptr: mylineInfoPtr ;
      thelineInfoRec: LineInfoRec ;
      cacheheight   : INTEGER ;
      linedraw      : INTEGER ;
      leftcol             : INTEGER ;


BEGIN
      savedClip := NewRgn;
      theWindow := WindowPtr(theEvent.message);
      pWindow := windowP(GetWRefCon(theWindow));
```

```
        GetPort(savedPort);
        SetPort(theWindow);
        GetClip(savedClip);                          { Get the old area }
        ClipAll(theWindow);

        BeginUpdate(theWindow);
                EraseRect(theWindow^.portRect);

                DrawControls(theWindow);
                DrawGrowIcon(theWindow);
                theTerm := GethTerm(theWindow);
                ClipRect(theTerm^^.viewRect);
                IF theTerm <> NIL THEN
                        TMUpdate(theTerm, theWindow^.visRgn);

        EndUpdate(theWindow);

        SetClip(savedClip);
        DisposeRgn(savedClip);
        SetPort(savedPort);
END;
```

Key events

```
PROCEDURE DoKey(theEvent : EventRecord);
VAR
        theKey : CHAR;
        processed : BOOLEAN;
        result : LONGINT;

BEGIN
        theTerm := GethTerm(FrontWindow);
{       IF theTerm <> NIL THEN      }
{               TMKey(theTerm,theEvent);    }

        theKey := CHAR(BAND(theEvent.message, charCodeMask));
        processed := FALSE;

        IF BAND(theEvent.modifiers, cmdKey) <> 0 THEN
                BEGIN
                result := MenuKey(theKey);
                IF HiWord(result) <> 0 THEN
                        BEGIN
                        processed := TRUE;
                        DoCommand(result);
                        END;
                IF NOT processed THEN
                        BEGIN
```

```
                IF theTerm <> NIL THEN
                        TMKey(theTerm, theEvent);
                END;
        END
    ELSE
        BEGIN
        IF theTerm <> NIL THEN
                TMKey(theTerm, theEvent);
        END;

END;
```

Mouse events

```
PROCEDURE DoClick(theEvent : EventRecord);
VAR
    thePart                 : INTEGER;

    theWindow           : WindowPtr;
    theWindowPeek       : WindowPeek;
    savedPort           : GrafPtr;
    result                  : LONGINT;
    theRect             : Rect;
    err                 :     OSErr;

BEGIN
    thePart := FindWindow(theEvent.where, theWindow);
    CASE thePart OF
        inGoAway:
            IF TrackGoAway(theWindow, theEvent.where) THEN
                DoClose;
        inDesk,
        inSysWindow:
            SystemClick(theEvent, theWindow);

        inZoomIn, inZoomOut:
            BEGIN
            IF theWindow = FrontWindow THEN
                IF TrackBox(theWindow, theEvent.where,
                thePart) THEN
                    BEGIN
                    GetPort(savedPort);
                    SetPort(theWIndow);
                    ClipAll(theWindow);

                    EraseRect(theWindow^.portRect);
                    ZoomWindow(theWIndow, thePart, FALSE);
                    ClipAll(theWindow);
```

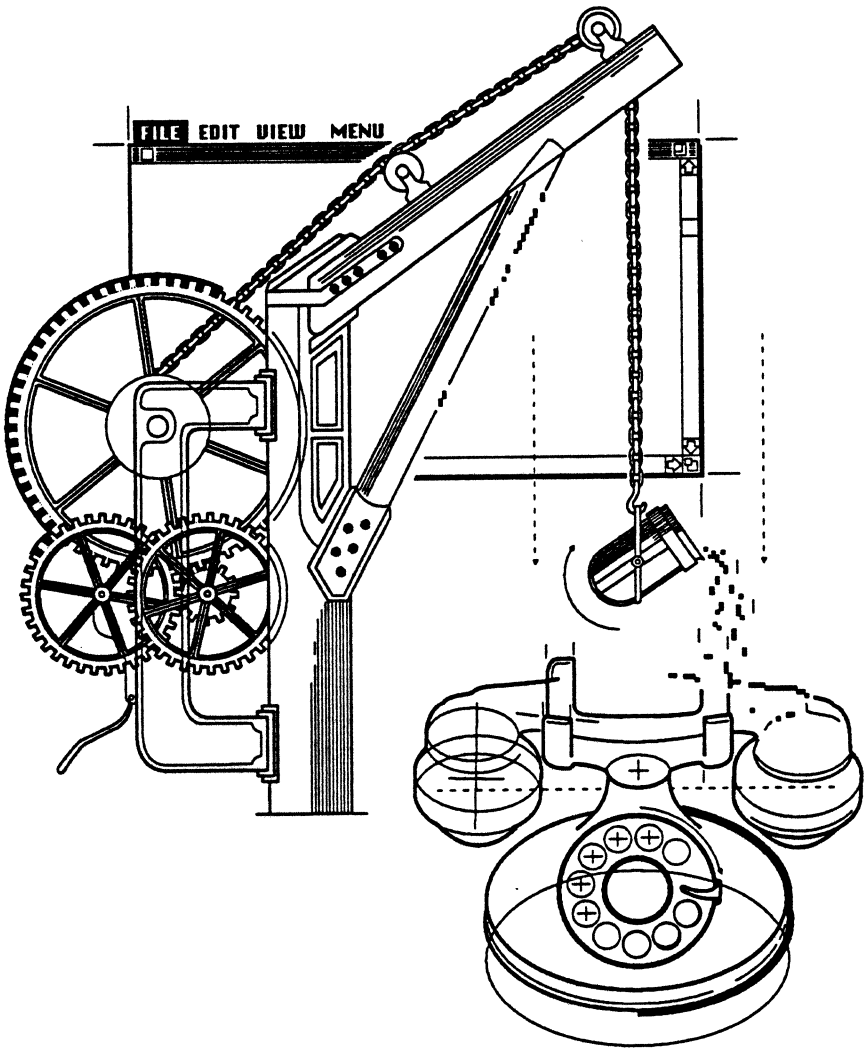
```
                        InvalRect(theWindow^.portRect);

                        ClipContent(theWindow);
                        SetPort(savedPort);
                        END;
                END;

        inMenuBar:
                BEGIN
                result := MenuSelect(theEvent.where);    { gee mr
                menu, what have we here? }

                DoCommand(result);          { chop chop chop }

                HiliteMenu(0);              { fix the menu bar }
                END;
        inGrow:
                BEGIN
                IF theWindow = scrapWindow THEN
                BEGIN
                        GrowScrap(theEvent.where);
                END
                ELSE
                        DoGrow(theWindow,theEvent.where);
                END;
        inDrag:         BEGIN
                        theRect := theWindow^.visRgn^^.rgnBBox;
                        DragWindow(theWindow, theEvent.where,
                        dragRect);
                END;
        inContent:
                IF theWindow <> FrontWindow THEN
                        SelectWindow(theWindow)
                ELSE BEGIN
                        theTerm := GethTerm(theWindow);
                        IF theTerm <> NIL THEN
                                TMClick(theTerm, theEvent);
                END; {of inContent}
        END; {case}
END;
```

Sample application main program loop

```
PROCEDURE MainLoop;
VAR
        theEvent : EventRecord;
        theWindow : WindowPtr;
        theWindowPeek : WIndowPeek;
        theControl : ControlHandle;
        savedPort : GrafPtr;
        theKey : CHAR;

        result : LONGINT;
        hFT: FTHandle;
        hConn: ConnHandle;
        hTerm: TermHandle;

BEGIN
        WHILE NOT done DO
        BEGIN
                SystemTask;

                DoIdle;
                IF WaitNextEvent(everyEvent,theEvent, 0, NIL) THEN
                BEGIN
                        IF theEvent.what = activateEvt THEN
                                dirtyMenu := TRUE;

                        hFT := IsFTEvent(theEvent);
                        IF hFT <> NIL THEN
                                FTEvent(hFT, theEvent)
                        ELSE
                        BEGIN
                                hConn := IsConnEvent(theEvent);
                                IF hConn <> NIL THEN
                                        CMEvent(hConn, theEvent)
                                ELSE
                                BEGIN
                                        hTerm := IsTermEvent(theEvent);
                                        IF hTerm <> NIL THEN
                                                TMEvent(hTerm, theEvent)
                                        ELSE
                                        BEGIN
                                                CASE theEvent.what OF
                                                        autoKey, keyDown:    { no
                                                        command-key equivalents
```

```
                                                    on a mac plus }
                                                          DoKey(theEvent);
                                                    mouseDown:

            DoClick(theEvent);
                                                    updateEvt:

            DoUpdate(theEvent);
                                                    app4Evt:

            DoResume(theEvent);
                                                    activateEvt:

            DoActivate(theEvent);
                                              END; {case}
                                          END; {is not TermEvent }
                                    END;{ is not CMEvent }
                              END; {is not FT Event }
                        END; {wne}

            IF dirtyMenu = TRUE THEN
                  UpdateMenus;
            END; {while not done}

END;
```

Chapter 3 **Connection Manager**



About this chapter

This chapter describes the Connection Manager, which is the Special K manager that allows applications to establish and maintain connections. This chapter starts out by describing some of the fundamental concepts about the Connection Manager. Then it describes the connection record, which is the most important data structure to the Connection Manager. After a detailed functional description of the routines that the Connection Manager provides, the chapter finishes with a summary you can use as a quick reference to routines and data structures.

Often refered to in this chapter is the term "your application", which is the application you are writing for the Macintosh, and which will implement communication services for users. Be careful not to confuse the services your application is requesting with the services that tools provide.

To use the Connection Manager, you need to be familiar with the concept of data connections, as well as the following:

- Resource Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Device Manager (see *Inside Macintosh*, Volumes: II, IV, V)

About the Connection Manager

By using Connection Manager routines, your application can implement basic connection services without having to take into account underlying connection protocols. Connection tools, which are discussed in Chapter 9: "How to Write a Connection Tool," are responsible for implementing connection services according to specific protocols.

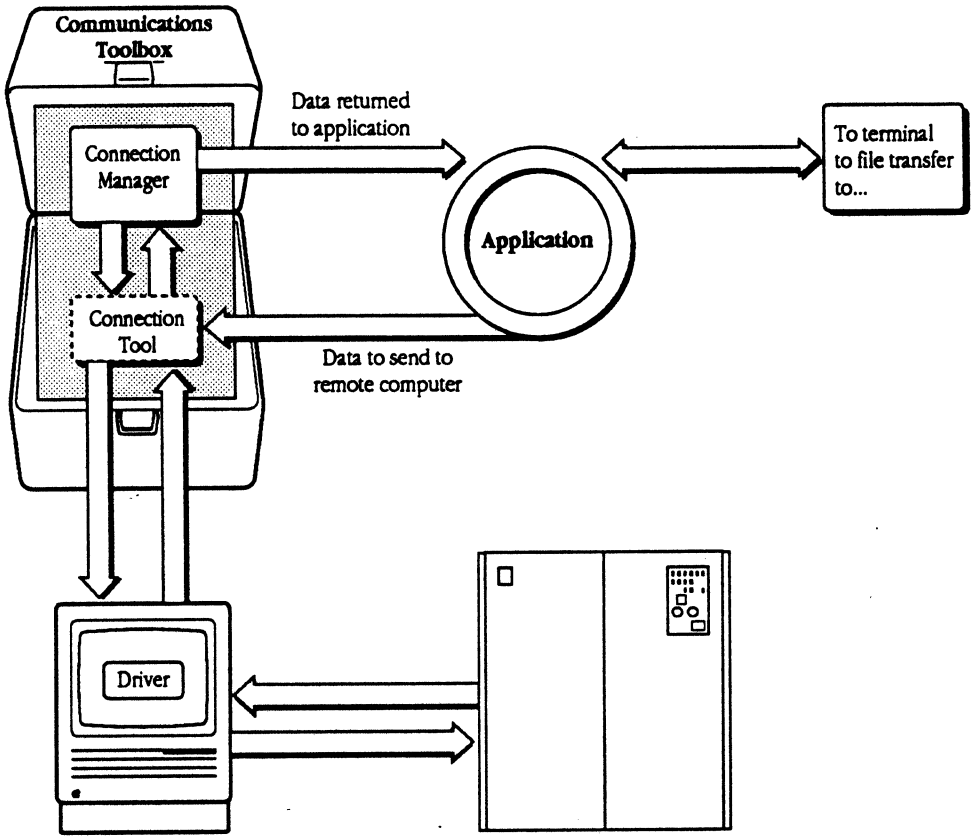
To the application, the Connection Manager provides a basic abstraction of a connection as being a two-way channel that carries data between two entities, or processes running on computer. These entities can be different processes running on the same CPU or one process running on a Macintosh and the other running on a mainframe (or any other type of computer).

Here's what happens inside the Connection Manager. An application makes a request of the Connection Manager when it needs it to do something, such as open a connection. The Connection Manager then sends this request on to one of the tools it manages. The tool, in turn, takes the request and executes the service according to the specifics of the connection protocol that is implemented on the data connection. Once the tool has finished, it passes back to the application any relevant parameters and return codes.

The data sent along the connection is represented in a stream, rather than a transaction-by-transaction basis. Flow control, error correction, error detection, and encapsulating the data into packets are not implemented with the connection, although a tool or application can provide these features. The Connection Manager does, however, provide function that tells if a connection is reliable (that is, it will transmit without errors and in the correct sequence).

Figure 3-1 shows the data flow into and out of the Connection Manager.

Figure 3-1: Data flow into and out of the Connection Manager.



The most important data structure maintained by the Connection Manager is the *connection record*, which is where all the specifics about a connection are stored. For example, the connection record might show that a connection is a direct serial port connection transmitting at 9600 baud.

Two reasons why the connection record is so important to the Connection Manager are that its existence allows for both protocol-independent routines and multiple instances of the same tool. Protocol-independent routines are what allow applications to use Connection Manager services without regard for the underlying communications protocols. In other words, when an application wants to read data from a remote entity, it tells the Connection Manager to read, and the Connection Manager figures out exactly how to implement a read on a given connection. Multiple instances of the same tool allows for the same tool to be used by different processes at the same time, like in a MultiFinder environment, or by different threads in a given application. The connection record is described in greater detail later in this chapter.

Besides providing basic connection routines, the Connection Manager includes routines that make it easy for applications to configure a connection tool, either through presenting the user with a dialog box or by interfacing directly with a scripting language. The Connection Manager also contains routines that make it easier for you to localize your applications into foreign languages.

You can use the Connection Manager in conjunction with other parts of Special K to create a communications application with file transfer and terminal emulation capabilities. Or, you can use

the Connection Manager from Special K, but substitute some other data transfer or terminal emulation service on top of Special K's Connection Manager. You can also write your own connection tool and add it to the Connection Manager. (This procedure is discussed in Chapter 8, "Fundamentals of Writing Your Own Tool.").

Connection channels: data, attention, and control

When data is sent along a connection there is a certain amount of overhead that sometimes accompanies it. This "extra" information could be a warning that the connection is about to go down or that the sending entity should slow its rate of transmitting data. Some connection protocols are designed in such a way that this sort information can be sent simultaneously with the data stream on a *channel*. The Connection Manager supports up to three channels on each connection—data, attention, and control—that can be thought of as three separate lines of communication between each entity. The data channel, however, is for all protocols the primary channel for transmitting information between entities. The other two channels are used by only some connection protocols.

When you design your application, keep in mind that some protocols support all three channels while others support only one (the data channel). Your application should be able to handle different connection tools such that users can change tools and still be able to use your program.

The connection record

The connection record contains both information that describes a connection, as well as pointers to Connection Manager internal data structures. The Connection Manager uses this information to "translate" the protocol-independent routines used by an application or tool into a service implemented according to a specified protocol. Most of the fields in the File Transfer record are filled in when an application calls `CMNew`, which is described later in this chapter.

Because the context for a given connection is maintained in a connection record, an application can communicate on more than one connection at the same time. All the application has to do is create a new connection record every time it initiates a new connection.

△ **Important** Your application, in order to be compatible with future releases of the Connection Manager, should not directly manipulate the fields of the connection record. The Connection Manager provides routines that applications and tools can use to change connection record fields. △

The connection record data structure

```
TYPE
      ConnHandle          =     ^ConnPtr;
      ConnPtr             =     ^ConnRecord;
      ConnRecord          =     RECORD
            procID        :     INTEGER;

            flags         :     LONGINT;
            errCode       :     CMErr;

            refCon        :     LONGINT;
            userData      :     LONGINT

            defProc       :     ProcPtr;

            config        :     Ptr;
            oldConfig     :     Ptr;

            reserved0     :     LONGINT;
            reserved1     :     LONGINT;
            reserved2     :     LONGINT;

            private       :     Ptr;

            bufferArray   :     Buffers;
            bufSizes      :     BufferSizes;

            mluField      :     LONGINT;

            asynchCount   :     BufferSizes;

      END;
```

procID

`procID` is the connection tool ID. This value is dynamically assigned by the Connection Manager.

flags

`flags` is a bit field that indicates certain specifics about a connection when the connection record is first created. The bit masks for `flags`.are:

```
CONST
        cmData          =       1;      { data channel available    }
        cmCntl          =       2;      { control channel avail     }
        cmAttn          =       4;      { attention channel avail   }

        cmDataClean     =       8;      { reliable data channel available }
        cmCntlClean     =       16;     { reliable control channel available }
        cmAttnClean     =       32;     { reliable attention channel available }

        cmNoMenus       =       64;     { don't display custom menus }
        cmQuiet         =       128     {don't display alert dialogs };
```

Your application can specify if the `CMNoMenus` or `CMQueit` bits should be on when it calls `CMNew` (which is discussed later in this chapter). The connection tool will set the rest of these bits

If the `cmData, cmCntl,` or `cmAttn` bits are set, a data, control, or attention channel is available for use. If the `cmDataClean, cmCntlClean,` or `cmAttnClean` bits are set, a reliable (error-free, in-order delivery) data, control, or attention circuit is available.

If the `cmNoMenus` bit is set, the connection tool will not display any custom menus. If the `cmQuiet` bit is set, the connection tool will not display any dialog boxes to alert the user of error conditions. These two bits are typically used to when interfacing with a scripting language.

errCode

`errCode` contains the last error encountered by the Connection Manager. Valid error codes are:

```
CONST
        cmRejected              =       1;
        cmFailed                =       2;
        cmTimeOut               =       3;
        cmNotOpen               =       4;
        cmNotClosed             =       5;
        cmNoRequestPending      =       6;
        cmNotSupported          =       7;
        cmNoTools               =       8;
```

refCon

`refCon` is a four-byte field for use by the application. In a multiple-connection record environment, `refCon` is used to distinguish one connection record from another.

userData

`userData` is a four-byte field that the application can use to store and access values for any purpose.

defProc

`defProc` is a procedure pointer to the main code resource of the connection tool that will implement the specifics of the connection protocol. The connection tool's main code resource is of type "`cdef`."

config

`config` is a pointer to a data block that is private to the connection tool. It can contain information like baud rate, parity, or handshaking for direct asynchronous connections, phone numbers for modem connections, or an NBP address for an AppleTalk connection, but this varies from tool to tool. The connection tool uses this record to store connection information. You can find a description of `config` later in this book in Chapter 8, "Fundamentals of Writing Your Own Tool." However, as an application developer, you don't need to be concerned with this field. All you need to know is that the connection tool, when selected, will fill in `config`. To see how this is done, read "How to Select a Connection Tool" on page nn.

oldConfig

`oldConfig` is a pointer to a data block that is private to the connection tool and contains an "old" version of the configuration block.

reserved0, reserved1, and reserved2

`reserved0`, `reserved1`, and `reserved2` are fields that are reserved for the Connection Manager.

▲ Warning: Do not alter the `reserved0`, `reserved1`, or `reserved2` fields in the connection record. ▲

private

`private` is a pointer to a data block that is private to the connection tool. Your application should not use this field.

bufferArray

bufferArray is a set of pointers to buffers for the data, control, and attention channels. These are the buffers that are used to read data in or write data out of the entity. These fields are the exclusive property of the connection tool. To have your application specify the size of these buffers it must specify the size it desires in desiredSizes, a parameter to the CMNew routine (which is discussed later in this chapter). To have the tool set the size of these buffers to tool-desired sizes, your application must pass a 0 in the desiredSizes parameter to CMNew. These buffers are the exclusive property of the connection tool and should not be used by an application. The data type for bufferArray is Buffers and is defined under the description of bufSizes.

bufSizes

bufSizes contains the actual sizes of the buffers and it too should not be manipulated directly by an application. The data type for bufSizes is BufferSizes, and are defined as:

```
TYPE
      Buffers              =      ARRAY[BufFields] OF Ptr;
      BufferSizes          =      ARRAY[BufFields] OF LONGINT;
```

The indexes of the arrays are the same and are:

```
CONST
      cmDataIn             =      0;
      cmDataOut            =      1;

      cmCntlIn             =      2;
      cmCntlOut            =      3;

      cmAttnIn             =      4;
      cmAttnOut            =      5;
```

mluField

mluField is a pointer to a private data structure that the Connection Manager uses when searching the data stream.

asyncCount

asyncCount is used by completion routines to indicate how many bytes were actually transmitted or received on a particular channel. Completion routines are discussed in more detail later in this chapter.

Connection Manager routines

This section describes the routines that tools and applications can use to access Connection Manager services. These routines are protocol-independent; your application does not need to be familiar with the specifics of a particular communications protocol in order to use the connection.

```
InitCM / nn                    CMGetProcID / nn
CMNew / nn                     CMDefault / nn
CMValidate / nn                CMChoose / nn
CMSetupPreFlight / nn          CMSetupSetup / nn
CMSetupFilter / nn             CMSetupItem / nn
CMSetupCleanup / nn            CMSetupPostFlight / nn
CMGetConfig / nn               CMSetConfig / nn
CMOpen / nn                    CMClose / nn
CMAbort / nn                   CMDispose / nn
CMIdle / nn                    CMListen / nn
CMAccept / nn                  CMRead / nn
CMWrite / nn                   SearchCallBack / nn
CMRemoveSearch / nn            CMClearSearch / nn
CMIOKill / nn                  CMStatus / nn
CMActivate / nn                CMResume / nn
CMEvent / nn                   CMMenu / nn
CMReset / nn                   CMBreak / nn
CMGetConnEnvirons / nn         CMIntlToEnglish / nn
CMEnglishToIntl / nn           CMSetRefCon / nn
CMGetRefCon / nn               CMSetUserData / nn
CMGetUserData / nn             CMGetVersion / nn
CMGetCMVersion / nn            MyCompletion / nn
```

Preparing to open a connection

Before your application can open a connection, it must first initialize the Connection Manager (`InitCM`), find out the `procID` of the tool it requires (`CMGetProcID`), create a connection record (`CMNew`), and then configure the connection tool (`CMChoose`).

`InitCM` Initializing the Connection Manager

`InitCM` initializes the Connection Manager. Your application should call this routine after calling the standard Macintosh Toolbox initialization routines. If your application uses either the Communications Resource Manager or the Special K Utilities, it should initialize them before initializing the Connection Manager.

Function

```
InitCM  :  CMErr;
```

Description `InitCM` returns an operating system error code if appropriate. If no tools are installed in the Connection Manager, it returns `cmNoTools`. Your application is responsible to check for the presence of the Communications Toolbox before calling this function.

`CMGetProcID` Getting current procID information

Your application should call `CMGetProcID` just before creating a new connection record to find out the `procID` of a tool.

Function

```
CMGetProcID(name: STR255): INTEGER;
```

Description `name` specifies a connection tool. If a connection tool exists with the specified `name`, its tool ID is returned. If `name` references a nonexistent connection tool, -1 is returned.

CMNew — Creating a connection record

Before your application can open a connection, it must first create a connection record so the Connection Manager knows what type of connection to establish. CMNew creates a new connection record, fills in the fields that it can based upon the parameters that were passed to it, and returns a handle to the new record in ConnHandle. The Connection Manager then loads the connection tool's main code resource and locks it. If memory constraints prevent a new connection record from being created, CMNew passes back NIL in ConnHandle.

Function

```
CMNew(theProcID : INTEGER; theFlags : LONGINT;
desiredSizes : BufferSizes; theRefCon : LONGINT;
theUserData : LONGINT) : ConnHandle;
```

Description

theProcID is dynamically assigned by the Connection Manager to tools at run time. Applications should not store procIDs in settings files. Instead, they should store tool names, which can be converted to procIDs with the CMGetProcID routine Your application should use the ID that CMGetProcID returns for theProcID.

theFlags is a bit field with the following masks:

```
CONST
      cmData          =      1;     { data channel reequest    }
      cmCntl          =      2;     { control channel request  }
      cmAttn          =      4;     { attention channel request }

      cmDataClean     =      8;     { reliable data channel request }
      cmCntlClean     =      16;    { reliable control channel request }
      cmAttnClean     =      32;    { reliable attention channel request }

      cmNoMenus       =      64;    { don't display custom menus }
      cmQuiet         =      128    { don't display alert dialogs };
```

theFlags represents a request from your application for a level of connection service. Your application can set only two of these bits, cmNoMenus and cmQuiet. If your application sets cmNoMenus, the connection tool will not display any custom menus. If your application sets cmQuiet, the connection tool will not display any dialogs to alert the user of error conditions. These two bits are typically used when interfacing with a scripting language.

The other bits are set by the connection tool. The level granted by the tool is returned in the flags field of the connection record, which is described on page nn, "The connection record data sturcture."

The bits of theFlags that are not shown in this manual are reserved for Apple Computer, Inc. Do not use them or your code may not work in the future.

desiredSizes specifies buffer sizes that your application requests for its read, write, control read, control write, attention read, and attention write channels. Your application can specify the sizes that it wants when it calls cmNew, but the connection tool might not provide the requested sizes. To have the tool set the size of these buffers, your application should pass 0 in this field.. These buffers become the exclusive property of the connection tool and should not be manipulated by the application in any way. The actual buffer sizes are kept in the bufSizes field of the connection record.

theRefCon and the UserData are fields for use by the application. In a multiple-connection environment theRefCon takes on special meaning in that it is used to distinguish among connection records.

CMDefault Initializing the configuration record

CMDefault fills the specified configuration record with the default configuration specified by the connection tool. This procedure is called automatically by CMNew when filling in the config and oldConfig fields in a new connection record.

Procedure

```
CMDefault(VAR theConfig: Ptr; procID: INTEGER; allocate:
BOOLEAN);
```

Description

If allocate is TRUE, the tool allocates space for theConfig in the current zone.

CMValidate Validating the configuration record

CMValidate validates the configuration and private data records of the connection record by comparing the fields in the connection record with the values that are specified in the connection tool. This routine is called by CMNew and CMSetConfig after they have created a new connection record to make sure that the the record contains values identical to those specified by the connection tool.

Function

```
CMValidate(hConn: ConnHandle): BOOLEAN;
```

Description

If the validation fails, the connection tool returns FALSE and fills the configuration record with default values by calling CMDefault.

CMChoose Configuring a connection tool

An application can configure a connection tool three ways. The easiest and most straightforward way is by calling the CMChoose routine. This routine presents the user with a dialog box similar to Figure 3-2. The second way an application can configure a connection tool is by presenting the user with a custom dialog box. This method is much more difficult and involves calling six routines. The routines are described under "Custom configuration of a connection" on page nn and some example code is provided in "Appendix B: Useful code samples" to help you implement this functionality. The third way your application can configure a connection tool is by interfacing directly with a scripting language. This method allows your application to bypass user interface elements.

To present the user with the standard tool-configuration dialog box, your application needs to call CMChoose, which will present the user with the dialog shown in Figure 3-2.

Function

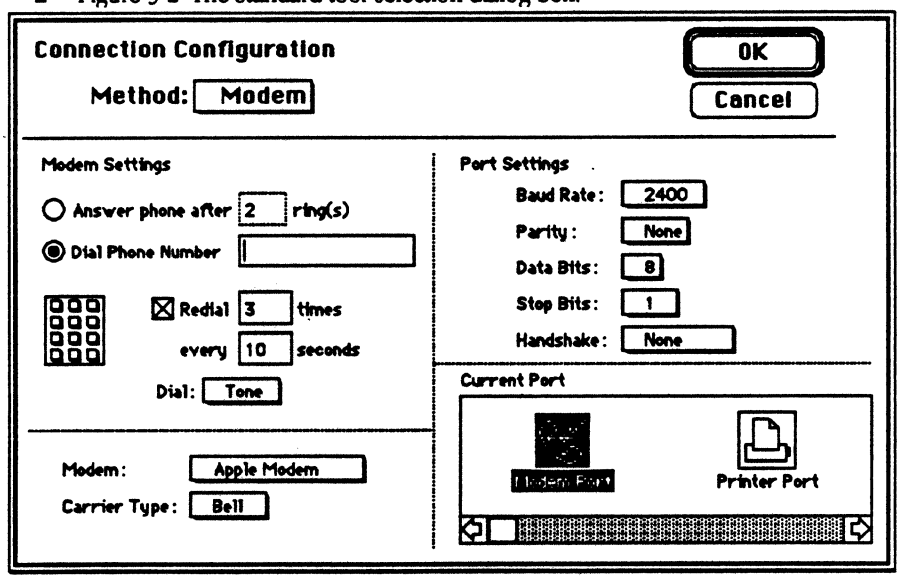
```
CMChoose(VAR hConn:ConnHandle; where: Point; idleProc:
ProcPtr): INTEGER;
```

Description

`where` is a point in global coordinates specifying the top left corner of where the dialog should appear. It is recommended that your application place the dialog as close to the top and left of the screen as possible.

`idleProc` is a procedure that the Connection Manager will automatically call every time your application loops through the setup dialog filter procedure.

■ Figure 3-2 The Standard tool-selection dialog box.



CMChoose will return one of the following values:

```
CONST
    chooseDisaster      =     -2;
    chooseFailed        =     -1;
    chooseAborted       =     0;
    chooseOKMinor       =     1;
    chooseOKMajor       =     2;
    chooseCancel        =     3;
```

chooseDisaster means that the CMChoose operation failed and destroyed the connection record.

chooseFailed means that the CMChoose operation failed and the connection record was not changed.

chooseAborted means that the user tried to change the connection while it was still open, thereby failing to complete the CMChoose operation.

chooseOKMinor means that the user selected OK in the dialog box, but did not change the connection tool being used.

chooseOKMajor means that the user selected OK in the dialog box and also changed the connection tool being used. The Connection Manager then destroys the old connection handle by calling CMDispose (the connection is closed down and all pending reads and writes are terminated), and a new connection handle is returned in hConn.

chooseCancel means that the user selected Cancel in the dialog box.

Custom configuration of a connection tool

To present the user with a custom tool-configuration dialog box, your application needs to call a series of six Connection Manger routines: CMSetupPreflight, CMSetupSetup, CMSetupItem, CMSetupFilter, CMSetupCleanup, and CMSetupPostflight. Using these routines is more involved than calling CMChoose, but they provide your application with much more flexibility. The code sample in Appendix, "Useful code samples" shows how an application calls these routines.

To build a list of available connection tools, use the routine CRMGetIndToolName, which is described in Chapter 6, "Communications Resource Manager."

CMSetupPreflight

CMSetupPreflight returns a handle to a dialog item list from the connection tool that the calling application will append to the configuration dialog box. (The calling application uses AppendDITL which is discussed in Chapter 7, "Special K Utilities.") CMSetupPreflight returns a value in magicCookie that should be passed into the other procedures.

Function

```
CMSetupPreflight(procID: INTEGER; VAR magicCookie:
LONGINT): Handle;
```

Description

procID is the ID for the connection tool that is being configured. Your application should get this value by using the CMGetProcID routine, which is discussed later in this chapter.

The connection tool can use CMSetupPreflight to allocate a block of private storage and to store the pointer to that block in magicCookie. This value, magicCookie, should be passed to the other routines that are used to setup the configuration dialog.

The refcon of the custom dialog box should point to a data structure in which the first two bytes are the tool procID and the next four bytes are magicCookie.

CMSetupSetup

CMSetupSetup tells the connection tool to set up controls (like radio buttons or check boxes) in the dialog item list returned by CMSetupPreflight.

Procedure

```
CMSetupSetup(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR magicCookie: LONGINT);
```

Description

procID is the ID for the connection tool that is being configured. Your application should get this value by using the CMGetProcID routine, which is discussed later in this chapter.

theConfig is a pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

magicCookie is the value returned from the CMSetupPreflight.

CMSetupFilter

CMSetupFilter should be called as a filter procedure prior to the standard modal dialog filter procedure for the configuration dialog box. This routine allows connection tools to filter events in the configuration dialog box.

Function

```
CMSetupFilter(procID: INTEGER; theConfig: Ptr;
count:INTEGER; theDialog: DialogPtr; VAR theEvent:
EventRecord; VAR theItem: INTEGER; VAR magicCookie:
LONGINT): BOOLEAN;
```

Description

count is the number of the first item in the dialog item list appended to the dialog box.

theConfig is a pointer to the configuration record for the tool being configured.

theDialog is the dialog box performing the configuration.

theEvent is the event record for which filtering is to take place.

theItem can return the appropriate item clicked on in the dialog box.

magicCookie is the value returned from CMSetupPreflight.

If the event passed in was handled, TRUE is returned. Otherwise, FALSE indicates that standard dialog filtering should take place.

CMSetupItem

CMSetupItem processes mouse events for controls in the custom configuration dialog box.

Procedure

```
CMSetupItem(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR item: INTEGER; VAR
magicCookie: LONGINT);
```

Description

procID is the ID for the connection tool being configured. Your application should get this value by using the CMGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

item is the item clicked on in the dialog box. This value can be modified and sent back.

magicCookie is the value returned from CMSetupPreflight.

CMSetupCleanup

CMSetupCleanup disposes of any storage allocated in CMSetupPreflight and performs any other clean-up operation. If your application needs to shorten a dialog box, it should do so after calling this routine.

Procedure

```
CMSetupCleanup(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR magicCookie: LONGINT);
```

Description

procID is the ID for the connection tool that is being configured. Your application should get this value by using the CMGetProcID routine, which is discussed later in this chapter.

theConfig is a pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

magicCookie is the value returned from CMSetupPreflight.

CMSetupPostflight

CMSetupPostflight will close the tool file if it is not being used by any other sessions.

Procedure

```
CMSetupPostflight(procID:INTEGER);
```

Description

procID is the ID for the connection tool that is being configured. Your application should get this value by using the CMGetProcID routine, which is discussed later in this chapter.

Scripting language interface

Your application does not have to rely on a user making selections from dialog boxes in order to configure a connection tool. CMGetConfig and CMSetConfig provide the function that your application needs to interface with a scripting language.

CMGetConfig

CMGetConfig returns a null-terminated string from the connection tool (an example of which is shown after the description of the next routine) containing tokens that fully describe the configuration of the connection record.

Function

```
CMGetConfig(hConn: ConnHandle): Ptr;
```

Description

If an an error occurs, CMGetConfig will return NIL. It is the responsibility of your application to dispose of Prt.

CMSetConfig

CMSetConfig passes a null-terminated string (an example of which is shown under "A sample null-terminated configuration string") to the connection tool for parsing. The string, which can be any length, is pointed to by thePtr, must contain tokens that describe the configuration of the connection record, and is parsed from left to right.

Function

```
CMSetConfig(hConn: ConnHandle; thePtr: Ptr): INTEGER;
```

Description

Items not recognized or relevant are ignored; this causes CMSetConfig to abort parsing the string and to return the character position where the error occurred. If parsing is successfully completed, CMSetConfig will return noErr. CMSetConfig may also return -1 to indicate a general problem with processing the configuration string..

The parsing operation is the responsibility of the individual tool.

A sample null-terminated configuration string

```
Baud 9600 BitsPerChar 8 Parity None StopBits 1 Port
ModemPort HandShake None HoldConnection False
RemindDisconnect False \0
```

Opening, using, and closing the connection

Once your application has performed the required tasks described above, it can then open and use a connection.

CMOpen — Opening a connection

CMOpen attempts to open a connection based on information in a connection record.

Function

```
CMOpen(hConn: ConnHandle; theAsync: BOOLEAN; completor:
ProcPtr; timeout: LONGINT): CMErr;
```

Description

hConn points to the connection record for the new connection.

theAsync specifies whether or not the opening request is asynchronous. If an asynchronous request is made, noErr is returned immediately.

completor specifies the completion routine to be called upon completion of the open request. Completion routines are discussed in greater detail later in this chapter under "About completion routines" on page nn.

timeout specifies a time period, in ticks, within which CMOpen must be completed before a cmTimeOut error is returned. For no timeout, use –1. For a single attempt to open the connection, use 0.

If no error occurs during the open attempt, CMOpen returns noErr. If the value returned is negative, an operating system error occurred. If the value returned is positive, a Connection Manager error occurred.

CMClose — Closing a connection

There are two ways to close a connection: CMClose and CMAbort.

Function

```
CMClose(hConn: ConnHandle; theAsync: BOOLEAN; completor:
ProcPtr; timeout: LONGINT; now: BOOLEAN): CMErr;
```

Description

theAsync specifies whether or not the close request is asynchronous. If an asynchronous request is made, noErr is immediately returned.

completor specifies the completion routine to be called upon completion of the close request. Completion routines are discussed in greater detail later in this chapter under "About completion routines" on page nn.

timeout specifies a time period, in ticks, within which the close must be completed before a cmTimeOut error will be returned. For no timeout, use –1. For a single try to close the connection, use 0.

CMAbort — Aborting a connection

CMAbort tells the Connection Manager to stop trying to open a pending asynchronous open request.

Function

```
CMAbort(hConn: ConnHandle): CMErr;
```

CMDispose — Disposing of a connection record

CMDispose disposes of the connection record and all associated data structures. It is up to the connection tool to decide wether or not to wait for all pending reads and writes to complete before closing and disposing of the connection.

Applications and tools need to distinguish between closing a connection with CMDispose and CMClose.

Procedure

```
CMDispose(hConn: ConnHandle);
```

CMIdle — Idle procedure

Your application should call CMIdle at least once every time it goes through its main event loop so that the connection tool can perform idle loop tasks.

Procedure

```
CMIdle(hConn: ConnHandle);
```

Description

hConn specifies the connection for which idle loop tasks are to be performed.

CMListen — Listening for incoming connection requests

CMListen "listens" for a connection request from another entity.

Function

```
CMListen(hConn: ConnHandle; theAsync: BOOLEAN; completor:
ProcPtr; timeout: LONGINT): CMErr;
```

Description

theAsync specifies whether or not the opening request is asynchronous. If an asynchronous request is made, noErr is returned immediately. If a synchronous request is made, CMListen stays in a "listen loop" until it receives the connection request.

`completor` specifies the completion routine that the Connection Manager calls after it is done listening for the connection request. Completion routines are discussed in greater detail later in this chapter under "About completion routines" on page nn.

`timeout` specifies a time period, in ticks, within which a connection request must be received before a `CMTimeOut` error is returned. For no timeout, use –1. For a single listen, use 0.

`CMAccept` Accepting or rejecting a connection request

`CMAccept` accepts or rejects an incoming connection request.

Function

```
CMAccept(hConn:ConnHandle; accept:BOOLEAN): CMErr;
```

Description

Typically, an application will perform some actions after a `CMListen`, the results of which determines whether or not to accept the request.

`CMIOKill` Stopping an asynchronous input/output request

`CMIOKill` terminates any pending input/output (I/O) requests on the specified subcircuit.

Function

```
CMIOKill(hConn: ConnHandle; which: INTEGER): CMErr;
```

Description

`which` indicates the subciruit, and can take on the following values:

```
CONST
    CMDataIn            =    0;
    CMDataOut           =    1;

    CMCntlIn            =    2;
    CMCntlOut           =    3;

    CMAttnIn            =    4;
    CMAttnOut           =    5;
```

`CMReset` Resetting the connection

`CMReset` causes the connection to be reset. The exact state to which the connetion is reset is dependent upon the connection protocol being implemented. All local read and write buffers are cleared.

Procedure `CMReset(hConn: ConnHandle);`

CMBreak — Sending breaks

`CMBreak` effects a break operation upon the connection. The exact effect of this operation depends upon the tool that is being used.

Procedure

```
CMBreak(hConn: ConnHandle; duration: LONGINT; theAsync:
BOOLEAN; completor: ProcPtr);
```

Description

`duration` specifies in ticks how long the connection tool should perform the break operation.

`completor` specifies the completion routine to be called upon completion of the break. Completion routines are discussed in greater detail later in this chapter under "About completion routines" on page nn.

CMStatus — Getting connection status information

`CMStatus` returns a variety of useful status information about a connection.

Function

```
CMStatus(hConn: ConnHandle; VAR sizes: BufferSizes; VAR
flags: LONGINT): CMErr;
```

Description

`sizes` is an array of six `LONGINT`s that contains the number of characters to be read or written in the data, control, and attention channels. The indexes of the array are:

```
CONST
    cmDataIn            =       0;
    cmDataOut           =       1;

    cmCntlIn            =       2;
    cmCntlOut           =       3;

    cmAttnIn            =       4;
    cmAttnOut           =       5;
```

`flags`, is a bit field with the following masks:

```
CONST
    cmStatusOpening             =       $0001;
    cmStatusOpen                =       $0002;
    cmStatusClosing             =       $0004;
    cmStatusDataAvail           =       $0008;
    cmStatusCntlAvail           =       $0010;
```

```
cmStatusAttnAvail          =      $0020;

cmStatusDRPend             =      $0040;
                        {data read pending}
cmStatusDWPend             =      $0080;
                        {data write pending}
cmStatusCRPend             =      $0100;
                        {cntl read pending}
cmStatusCWPend             =      $0200;
                        {cntl write pending}
cmStatusARPend             =      $0400;
                        {attn read pending}
cmStatusAWPend             =      $0800;
                        {attn write pending}

cmStatusBreakPending       =      $1000;
cmStatusListenPend         =      $2000;
```

CMGetConnEnvirons Getting the connection environment

CMGetConnEnvirons returns the connection environment record for the connection specified by ConnHandle.

Function

```
CMGetConnEnvirons (hConn : ConnHandle; VAR theEnvirons :
ConnEnvironRec) : OSErr;
```

Description

CMGetConnEnvirons returns an appropriate operating system error (envVersTooBig) if the version requested is not available.

theEnvirons should contain the version requested in the version field of ConnEnvironRec, which for version 0 is:

```
TYPE
    ConnEnvironRecPtr          =      ^ConnEnvironRec;
    ConnEnvironRec             =      PACKED RECORD;
          version              =      INTEGER;
          baudRate             =      LONGINT;
          dataBits             =      INTEGER;
          channels             =      INTEGER;
          swFlowControl        =      BOOLEAN;
          hwFlowControl        =      BOOLEAN;
          eomAvailable         =      BOOLEAN;
          reserved             =      BOOLEAN;
END;
```

The connection tool is responsible for filling in the fields of ConnEnvironRec with either a value (in fields for which it has a valid value to supply) or 0.

Reading and writing data

The Connection Manager provides routines to read and write data from a buffer. Your application can also use the Connection Manager routine that reads data, `CMRead`, to search the incoming data stream for a specified pattern of bytes, the method for which is discussed under "Data Stream Searching."

`CMRead` Reading data

`CMRead` reads data into a block of memory. Your application can not queue multiple reads for the same channel on the same connection. However, your application can have both a pending read and a pending write can on the same channel at the same time.

Function

```
CMRead(hConn: ConnHandle; theBuffer: Ptr; VAR toRead:
LONGINT; theChannel: INTEGER; theAsync: BOOLEAN;
completor: ProcPtr; timeout: LONGINT; VAR theEOM:
BOOLEAN): CMErr;
```

Description

`theBuffer` specifies the buffer into which the connection tool should read data.

`toRead` specifies the number of bytes to be read.

`theChannel` specifies the channel on which reading is to take place; acceptable values are:

```
CONST
    cmData          =    1;
    cmCntl          =    2;
    cmAttn          =    4;
```

`theAsync` specifies whether or not the request is asynchronous. If an asynchronous request is made, `noErr` is returned immediately.

`completor` specifies the completion routine to be called upon completion of the read or write request. Completion routines are discussed in greater detail later in this chapter under "About completion routines" on page nn.

`timeout` specifies a time period, in ticks, within which the read must completed before a timeout error will occur. For no timeout, use –1. For a single read attempt, use 0.

theEOM indicates whether or not an end-of-message indicator was received. An end-of-message indicator needs to be supported by the particular communications protocol being used; if an end-of-message indicator is not supported by a connection protocol, your applicaiton should ignore this indicator.

CMWrite Writing data

CMWrite writes data from block of memory. Your application can not queue multiple writes for the same channel on the same connection. However, your application can have both a pending read and a pending write can on the same channel at the same time

Function

```
CMWrite(hConn: ConnHandle; theBuffer: Ptr; VAR toWrite:
LONGINT; theChannel: INTEGER; theAsync: BOOLEAN;
completor: ProcPtr; timeout: LONGINT; theEOM: BOOLEAN):
CMErr;
```

Description

theBuffer specifies the buffer from which the connection tool should get the data to write.

toWrite specifies the number of bytes to be written.

theChannel specifies the channel on which writing is to take place; acceptable values are shown below.

```
CONST
    CMData              =       1;
    CMCntl              =       2;
    CMAttn              =       4;
```

theAsync specifies whether or not the request is asynchronous. If an asynchronous request is made, noErr is returned immediately.

completor specifies the completion routine to be called upon completion of the write request. Completion routines are discussed in greater detail later in this chapter under "About completion routines."

timeout specifies a time period, in ticks, within which the write must completed before a timeout error will occur. For no timeout, use –1. For a single write attempt, use 0.

theEOM indicates whether or not end-of-message indicator should be sent. An end-of-message indicator needs to be supported by the particular communications protocol being used; if an end-of-message indicator is not supported by a connection protocol, your applicaiton should ignore this indicator.

CMAddSearch Data stream searching

When a tool or application is reading in data with CMRead, you can have the stream searched for one or more patterns of bytes. To perform the search, the Connection Manager needs to be given information such as the connection on which the data stream is coming in and the sequences of bytes for which to look. CMAddSearch tells the Connection Manager to perform the search, passing it search-specific information as well. Each time your application calls CMAddSearch, the Connection Manager will search for another sequence of bytes.

Function

```
CMAddSearch(hConn: ConnHandle; theString: STR255; flags:
INTEGER; callBack: ProcPtr): LONGINT;
```

Description

flags is a field that describes the search to be performed. The appropriate values are:

```
CONST
    CMSearchNoDiacrit   =     1;     { ignore diacriticals }
    CMSearchNoCase      =     2;     { ignore case }
```

callBack is a pointer to a routine the Connection Manager will call during CMRead in the event that a match is found. The calling conventions for the callBack procedure is shown under "What to do when a match is found." The value that is returned by CMAddSearch is a search reference number that is used by the CMRemoveSearch routine (described under "Stopping the data stream search"). If CMAddSearch returns -1, the search was not successfully added.

SearchCallBack What to do when a match is found

The Connection Manager will pass control to a search call-back procedure in the event that a match is found in the incoming data stream.

Procedure

```
SearchCallBack(hConn: ConnHandle; matchPtr: Ptr; refNum:
LONGINT);
```

Description

matchPtr points to the last matched character in the read buffer.

SearchCallBack returns a reference number, refNum, that is used to distinguish which sequence of bytes was found in the event that more than one search was taking place. refnum is the same value returned from CMAddSearch.

`CMRemoveSearch` Stopping the data stream search

`CMRemoveSearch` removes a search with the specified reference number for the specified connection record.

Procedure

```
CMRemoveSearch(hConn: ConnHandle; refNum: LONGINT);
```

Description `refnum` is the same value returned from `CMAddSearch`.

`CMClearSearch` Clearing all searches

`CMClearSearch` removes all searches associated with the specified connection record.

Procedure

```
CMClearSearch(hConn: ConnHandle);
```

Handling events

The Connection Manager event processing routines provide useful extensions to the Macintosh Toolbox Event Manager. There is example code in Appendix B, "Useful code samples" that shows how an application can determine if an event needs to be handled by one of these procedures.

`CMActivate` Activate events

`CMActivate` processes an activate or deactivate event (for instance, installing or removing a custom tool menu) for a window that the connection is associated with.

Procedure

```
CMActivate(hConn: ConnHandle; act: BOOLEAN);
```

Description If `act` is `TRUE`, an activate event is to be processed. Otherwise, a deactivate event is to be processed.

CMResume — **Resume events**

CMResume processes a resume or suspend for a window that the connection is associated with.

Procedure

```
CMResume(hConn: ConnHandle; res: BOOLEAN);
```

Description If `res` is `TRUE`, then a resume event is to be processed.

CMMenu — **Menu events**

Your application should call CMMenu in response to a selection from a menu that is installed by the connection tool.

Function

```
CMMenu(hConn: ConnHandle; menuID: INTEGER; item: INTEGER):
BOOLEAN;
```

Description CMMenu returns `FALSE` if the menu item was not handled by the connection tool. CMMenu returns `TRUE` if the connection tool was able to handle the menu item.

CMEvent — **Other events**

CMEvent is called only in response to receiving an event for a window that is associated with the connection; an example of such a window is a status dialog box that is created by the connection tool.

Procedure

```
CMEvent(hConn: ConnHandle; theEvent: EventRecord);
```

Windows (or dialog boxes) that are associated with the Connection Manager should have a connection record handle stored in the `refCon` field for `windowRecord`.

Localizing strings

Special K provides two routines that make it easier to localize strings.

CMIntlToEnglish

CMIntlToEnglish converts a configuration string, which is pointed to by inputPtr, to an American English configuration string that is pointed to by outputPtr.

Function

```
CMIntlToEnglish(hConn: ConnHandle; inputPtr: Ptr; VAR
outputPtr: Ptr; language: INTEGER): OSErr;
```

Description

language specifies the language from which the string is to be converted.

The connection tool allocates space for outputPtr.

If the language specified is not supported, noErr is still returned, but outputPtr is NIL.

The function returns an operating system error code if any internal errors occur.

CMEnglishToIntl

CMEnglishToIntl converts an English configuration string, which is pointed to by inputPtr, to a configuration string that is pointed to by outputPtr.

Function

```
CMEnglishToIntl(hConn: ConnHandle; inputPtr: Ptr; VAR
outputPtr: Ptr; language: INTEGER): OSErr;
```

Description

language specifies the language to which the string is to be converted.

The connection tool allocates space for outputPtr.

If the language specified is not supported, noErr is still returned, but outputPtr is NIL.

The function returns an operating system error code if any internal errors occur.

Miscellaneous routines

CMSetRefCon

CMSetRefCon sets the connection record's reference constant to the specified value.

Procedure `CMSetRefCon(hConn: ConnHandle; rC: LONGINT);`

CMGetRefCon

`CMGetRefCon` returns the connection record's reference constant.

Function `CMGetRefCon(hConn: ConnHandle): LONGINT;`

CMSetUserData

`CMSetUserData` sets the connection record's `userData` field to the specified value. It is very important that your application uses this routine to change the value of the `userData` field instead of changing it directly.

Procedure `CMSetUserData(hConn: ConnHandle; uD: LONGINT);`

CMGetUserData

`CMGetUserData` returns the connection record's `userData` field.

Function `CMGetUserData(hConn: ConnHandle): LONGINT;`

CMGetVersion

`CMGetVersion` returns a handle to a relocatable block that contains the information that is in the connection tool's "`vers`" resource with ID=1. This handle is *not* a resource handle.

Function `CMGetVersion(hConn:ConnHandle): Handle;`

CMGetCMVersion

`CMGetCMVersion` returns the version of the Connection Manager being used.

Function `CMGetCMVersion: INTEGER;`

About completion routines

This section describes the syntax and conventions that a completion routine in your application should follow.

`MyCompletion` Writing a completion routine

The completion routine has the same restrictions as standard device manager completion routines. For example, allocating memory is not allowed.

Procedure `MyCompletion(hConn: ConnHandle);`

Description When the Connection Manager passes control to `MyCompletion`, A0 points to the parameter block, if one is available. If no parameter block is available, A0 is `NIL`. If the completion routine cannot be given the connection handle, `hConn` is `NIL`.

Also at the time the Connection Manager calls `MyCompletion`, the appropriate part of the `asynchCount` field in the connection record should be filled with appropriate values.

Summary

Connection Manager routines	*see page*
`CMAbort(hConn: ConnHandle): CMErr;`	nn
`CMAccept(hConn:ConnHandle; accept:BOOLEAN): CMErr;`	nn
`CMActivate(hConn: ConnHandle; act: BOOLEAN);`	nn
`CMAddSearch(hConn: ConnHandle; theString: STR255; flags: INTEGER; callBack: ProcPtr): LONGINT;`	nn
`CMBreak(hConn: ConnHandle; duration: LONGINT; theAsync: BOOLEAN; completor: ProcPtr);`	nn
`CMChoose(VAR hConn:ConnHandle; where: Point; idleProc:ProcPtr): INTEGER;`	nn
`CMClearSearch(hConn: ConnHandle);`	nn
`CMClose(hConn: ConnHandle; theAsync: BOOLEAN; completor: ProcPtr; timeout: LONGINT; now: BOOLEAN): CMErr;`	nn
`CMDefault(VAR theConfig: Ptr ; procID: INTEGER; allocate: BOOLEAN);`	nn
`CMDispose(hConn: ConnHandle);`	nn
`CMEnglishToIntl(hConn: ConnHandle; inputPtr: Ptr; VAR outputPtr: Ptr; language: INTEGER): INTEGER;`	nn
`CMEvent(hConn: ConnHandle; theEvent: EventRecord);`	nn
`CMGetCMVersion: INTEGER;`	nn
`CMGetConfig(hConn: ConnHandle): Ptr;`	nn
`CMGetConnEnvirons (hConn : ConnHandle; VAR theEnvirons : ConnEnvironRec) : OSErr;`	nn
`CMGetConnName(procID: INTEGER; VAR name: STR255);`	nn
`CMGetProcID(name: STR255): INTEGER;`	nn

```
CMGetRefCon(hConn: ConnHandle): LONGINT;                          nn

CMGetUserData(hConn: ConnHandle): LONGINT;                        nn

CMGetVersion(hConn:ConnHandle):Handle;                            nn

CMIdle(hConn: ConnHandle);                                        nn

CMIntlToEnglish(hConn: ConnHandle; inputPtr: Ptr; VAR             nn
              outputPtr: Ptr; language: INTEGER):
              INTEGER;

CMIOKill(hConn: ConnHandle; which: INTEGER): CMErr;               nn

CMListen(hConn: ConnHandle; theAsync: BOOLEAN;                    nn
              completor: ProcPtr; timeout: LONGINT):
              CMErr;

CMMenu(hConn: ConnHandle; menuID: INTEGER; item:                  nn
              INTEGER): BOOLEAN;

CMNew(theProcID: INTEGER; theFlags: LONGINT;                      nn
              desiredSizes: BufferSizes; theRefCon:
              LONGINT; theUserData: LONGINT):
              ConnHandle;

CMOpen(hConn: ConnHandle; theAsync: BOOLEAN;                      nn
              completor: ProcPtr; timeout: LONGINT):
              CMErr;

CMRead(hConn: ConnHandle; theBuffer: Ptr; VAR toRead:             nn
              LONGINT; theChannel: INTEGER; theAsync:
              BOOLEAN; completor: ProcPtr; timeout:
              LONGINT; VAR theEOM: BOOLEAN): CMErr;

CMRemoveSearch(hConn: ConnHandle; refNum: LONGINT);               nn

CMReset(hConn: ConnHandle);                                       nn

CMResume(hConn: ConnHandle; res: BOOLEAN);                        nn

CMSetConfig(hConn: ConnHandle; thePtr: Ptr): INTEGER;             nn
```

```
CMSetRefCon(hConn: ConnHandle; rC: LONGINT);                    nn

CMSetupCleanup(procID: INTEGER; theConfig: Ptr; count:          nn
              INTEGER; theDialog: DialogPtr; VAR
              magicCookie: LONGINT);

CMSetupFilter(procID: INTEGER; theConfig: Ptr;                  nn
              count:INTEGER; theDialog: DialogPtr; VAR
              theEvent: EventRecord; VAR theItem:
              INTEGER; VAR magicCookie: LONGINT):
              Boolean;

CMSetupItem(procID: INTEGER; theConfig: Ptr; count:             nn
              INTEGER; theDialog: DialogPtr; VAR item:
              INTEGER; VAR magicCookie: LONGINT);

CMSetupPostflight(procID:INTEGER);                              nn

CMSetupPreflight(procID: INTEGER; VAR magicCookie:              nn
              LONGINT): Handle;

CMSetupSetup(procID: INTEGER; theConfig: Ptr; count:            nn
              INTEGER; theDialog: DialogPtr; VAR
              magicCookie: LONGINT);

CMSetUserData(hConn: ConnHandle; uD: LONGINT);                  nn

CMStatus(hConn: ConnHandle; VAR sizes: BufferSizes;             nn
              VAR flags: LONGINT): CMErr;

CMValidate(hConn: ConnHandle): BOOLEAN;                         nn

CMWrite(hConn: ConnHandle; theBuffer: Ptr; VAR                  nn
              toWrite: LONGINT; theChannel: INTEGER;
              theAsync: BOOLEAN; completor: ProcPtr;
              timeout: LONGINT; theEOM: BOOLEAN):
              CMErr;

InitCM:  CMERR                                                  nn
```

Routines in your application ***see page***

```
SearchCallBack(hConn: ConnHandle; matchPtr: Ptr;
              refNum: LONGINT);

MyCompletion(hConn: ConnHandle);
```

Data Types

Connection Record

```
TYPE
      ConnHandle          =     ^ConnPtr;
      ConnPtr             =     ^ConnRecord;
      ConnRecord          =     RECORD
            procID        :     INTEGER;

            flags         :     LONGINT;
            errCode       :     CMErr;

            refCon        :     LONGINT;
            userData      :     LONGINT

            defProc       :     ProcPtr;

            config        :     Ptr;
            oldConfig     :     Ptr;

            reserved0     :     LONGINT;
            reserved1     :     LONGINT;
            reserved2     :     LONGINT;

            private       :     Ptr;

            bufferArray   :     Buffers;
            bufSizes      :     BufferSizes;

            mluField      :     LONGINT;

            asynchCount   :     BufferSizes;

      END;
```

Connection Environment Record

```
TYPE
      ConnEnvironRecPtr          =      ^ConnEnvironRec;
      ConnEnvironRec             =      PACKED RECORD;
            version              =      INTEGER;
            baudRate             =      LONGINT;
            dataBits             =      INTEGER;
            channels             =      INTEGER;
            swFlowControl        =      BOOLEAN;
            hwFlowControl        =      BOOLEAN;
            eomAvailable         =      BOOLEAN;
            reserved             =      BOOLEAN;
END;
```

Data structures

dataBuffer

```
TYPE
      dataBufferPtr        =      ^dataBuffer;
      dataBuffer           =      RECORD
            thePtr         :      Ptr;
            count          :      LONGINT;
            channel        :      INTEGER;
            flags          :      BOOLEAN;
      END;
```

CompletorRecord

```
      CompletorPtr         =      ^CompletorRecord;
      CompletorRecord      =      RECORD
            async          :      BOOLEAN;
            completionRoutine
                           :      ProcPtr;
      END;
```

SetupStruct

```
      SetupPtr             =      ^SetupStruct;
      SetupStruct          =      RECORD
            theDialog      :      DialogPtr;
            count          :      INTEGER;
            theConfig      :      Ptr;
            procID         :      INTEGER
      END;
```

Constants

Connection record flags bit masks

```
CONST
      cmData          =      1;     { data channel available    }
      cmCntl          =      2;     { control channel avail     }
      cmAttn          =      4;     { attention channel avail   }

      cmDataClean     =      8;     { reliable data channel available }
      cmCntlClean     =      16;    { reliable control channel available }
      cmAttnClean     =      32;    { reliable attention channel available }

      cmNoMenus       =      64;    { don't display custom menus }
      cmQuiet         =      128    {don't display alert dialogs };
```

Buffers

```
      cmDataIn              =      0;
      cmDataOut             =      1;

      cmCntlIn              =      2;
      cmCntlOut             =      3;

      cmAttnIn              =      4;
      cmAttnOut             =      5;
```

Search flags

```
      CMSearchNoDiacrit     =      1;     { ignore diacriticals }
      CMSearchNoCase        =      2;     { ignore case }
```

Values returned by CMChoose

```
CONST
{     Choose return values }
      chooseDisaster        =      -2;
      chooseFailed          =      -1;
      chooseAborted         =      0;
      chooseOKMinor         =      1;
      chooseOKMajor         =      2;
      chooseCancel          =      3;
```

Connection status flags

```
cmStatusOpening             =       $0001;
cmStatusOpen                =       $0002;
cmStatusClosing             =       $0004;
cmStatusDataAvail           =       $0008;
cmStatusCntlAvail           =       $0010;
cmStatusAttnAvail           =       $0020;

cmStatusDRPend              =       $0040;
                            {data read pending}
cmStatusDWPend              =       $0080;
                            {data write pending}
cmStatusCRPend              =       $0100;
                            {cntl read pending}
cmStatusCWPend              =       $0200;
                            {cntl write pending}
cmStatusARPend              =       $0400;
                            {attn read pending}
cmStatusAWPend              =       $0800;
                            {attn write pending}

cmStatusBreakPending        =       $1000;
cmStatusListenPend  =       $2000;
```

Error codes

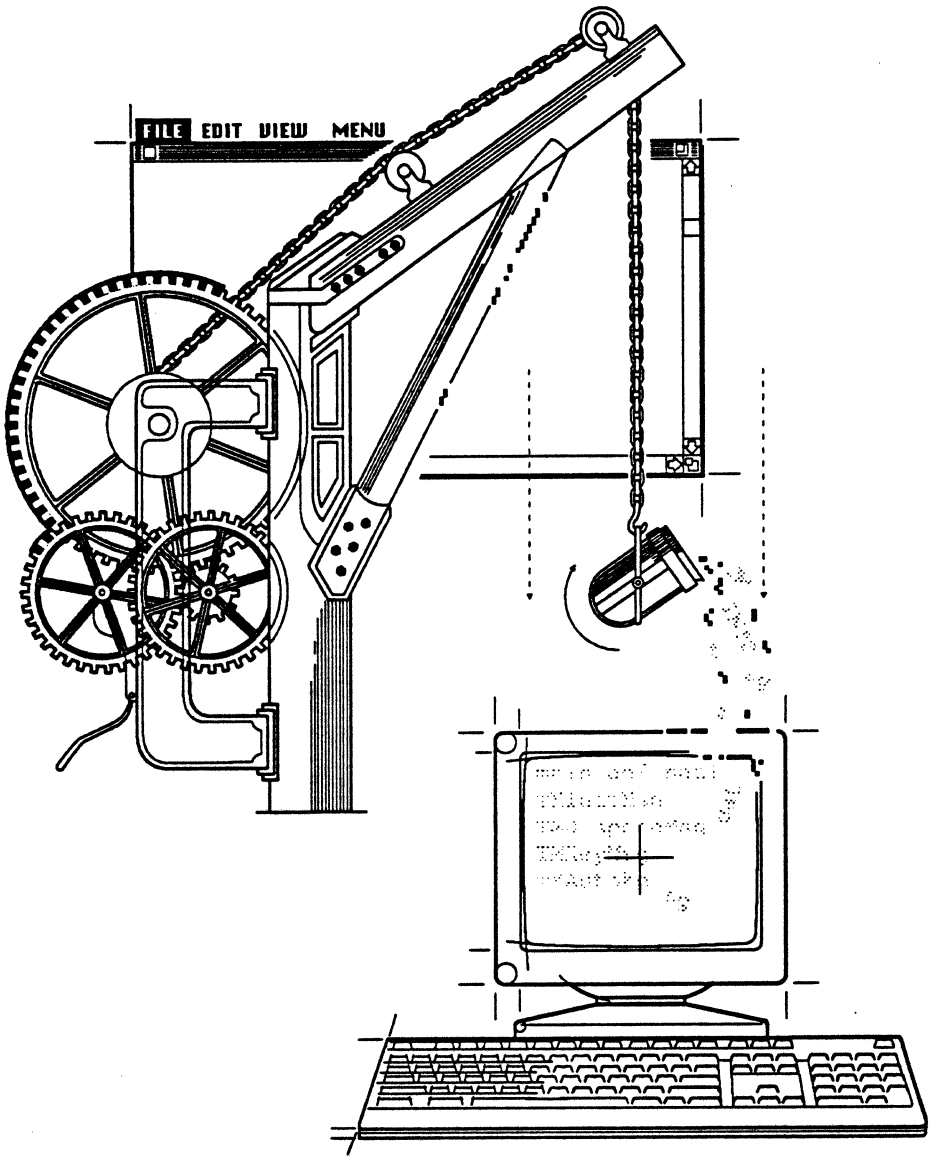
```
CONST
        cmRejected          =       1;
        cmFailed            =       2;
        cmTimeOut           =       3;
        cmNotOpen           =       4;
        cmNotClosed         =       5;
        cmNoRequestPending  =       6;
        cmNotSupported      =       7;
        cmNoTools           =       8;
```

Connection Manager routine selectors

```
CMAbort             .EQU    271         CMAccept            .EQU    269

CMActivate          .EQU    275         CMAddSearch         .EQU    294

CMBreak             .EQU    293         CMChoose            .EQU    292

CMClearSearch       .EQU    296         CMClose             .EQU    270

CMDefault           .EQU    280         CMDispose           .EQU    265
```

```
CMEnglishToIntl     .EQU    287         CMEvent             .EQU    298
CMGetCMVersion      .EQU    289         CMGetConfig         .EQU    284
CMGetConnEnvirons   .EQU    300         CMGetProcID         .EQU    263
CMGetRefCon         .EQU    259         CMGetToolName       .EQU    262
CMGetUserData       .EQU    261         CMGetVersion        .EQU    288
CMIdle              .EQU    266         CMIntlToEnglish     .EQU    286
CMIOKill            .EQU    297         CMListen            .EQU    268
CMMenu              .EQU    277         CMNew               .EQU    264
CMOpen              .EQU    267         CMRead              .EQU    273
CMRemoveSearch      .EQU    295         CMReset             .EQU    278
CMResume            .EQU    276         CMSetConfig         .EQU    285
CMSetRefCon         .EQU    258         CMSetupCleanup      .EQU    283
CMSetupFilter       .EQU    290         CMSetupItem         .EQU    282
CMSetupPostflight   .EQU    299         CMSetupPreflight    .EQU    291
CMSetupSetup        .EQU    281         CMSetUserData       .EQU    260
CMStatus            .EQU    272         CMValidate          .EQU    279
CMWrite             .EQU    274         InitCM              .EQU    257
```

Chapter 4 **Terminal Manager**



About this chapter

This chapter describes the Terminal Manager, which is the Special K manager that allows applications to perform terminal emulations independent of specific terminal characteristics. This chapter starts out by describing fundamental concepts about the Terminal Manager. Then it describes both the terminal emulation window and the terminal record, which is the most important data structure to the Terminal Manager. After a detailed functional description of the routines the Terminal Manager provides, this chapter finishes by describing the routines that need to be in your application. There is a summary at the very end of this chapter that you can use as a quick reference to routines and data structures.

Often refered to in this chapter is the term "your application", which is the application you are writing for the Macintosh, and which will implement communication services for users. Be careful not to confuse the services your application is requesting with the services that tools provide.

To use terminal tools in an application, you need to be familiar with general terminal emulation topics, as well as:

- Resource Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- QuickDraw (see *Inside Macintosh*, Volumes: I, V)
- Event Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Scrap Manager (see *Inside Macintosh*, Volume I)
- Dialog Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Connection Manager (see in this manual Chapter 3, "Connection Manager")

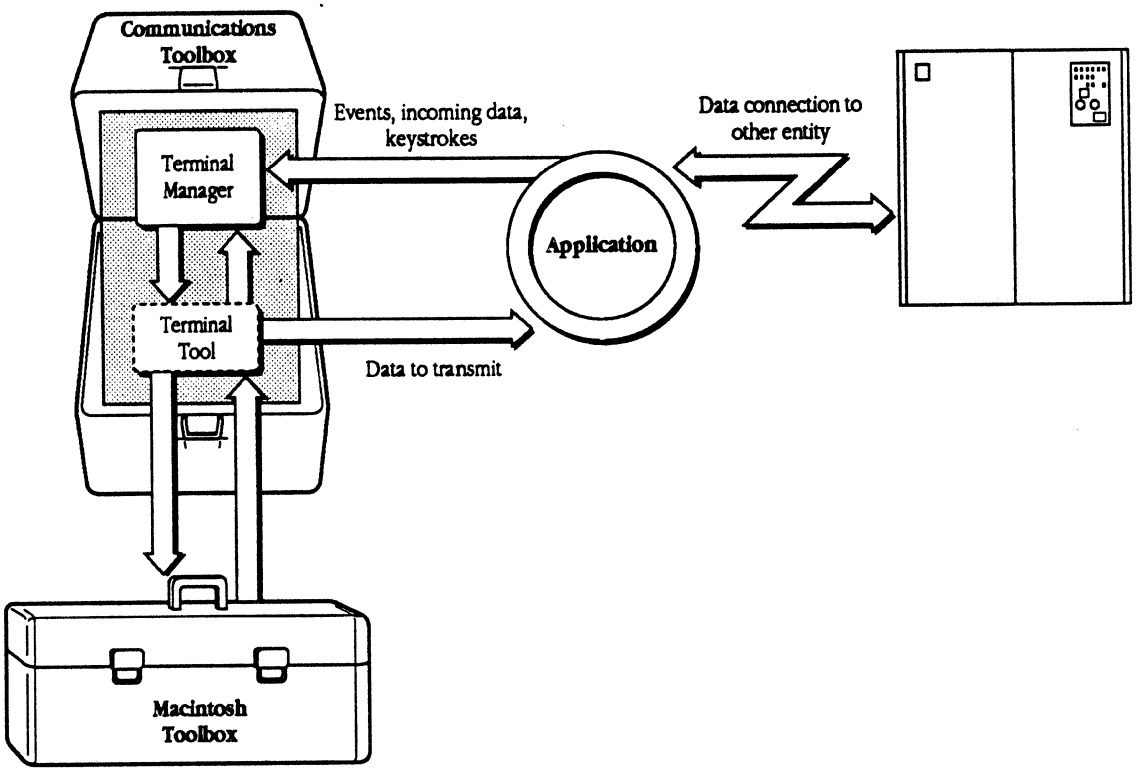
About the Terminal Manager

By using Terminal Manager routines, your application can implement terminal emulations without having to take into account specific terminal characteristics. Terminal tools, which are discussed in Chapter 10, "How to Write a Terminal Tool," are responsible for implementing the specifics of a terminal emulation.

To the application, the Terminal Manager provides a basic abstraction of a terminal emulation as being an interaction between the Macintosh and a host computer. This abstraction is best described with an example. Suppose your application needs to tell a mainframe at the other end of an existing connection that the user has typed the letter "a". Your application will first detect that the user has pressed a key and pass this event on to the Terminal Manager by calling the `TMKey` routine. Then, the Terminal Manager will pass this event on to the terminal tool, which had previously been selected. The terminal tool will figure out what the appropriate value is to transmit for "a" and send this value out on the connection. The example, of course, is a very simple one. But it is meant to give you a high-level feel for what goes on inside the Terminal Manager. The rest of this chapter goes into much more detail.

Figure 4-1 shows the data into and out of the Terminal Manager.

Figure 4-1: Data flow in and out of the Terminal Manager.



The most important data structure maintained by the Terminal Manager is the *terminal record*, which is where all the specifics about a terminal emulation are stored. For example, the terminal record might show that your application is emulating a VT320 terminal, and that the Terminal Manager should try to cache the terminal window before clearing it.

Two reasons why the terminal record is so important to the Terminal Manager are that its existence allows for both protocol-independent routines and multiple instances of the same tool. Protocol-independent routines are what allow applications to use Terminal Manager services without regard for the underlying communications protocols. In other words, when an application wants to transmit a keystroke to a host computer, it tells the Terminal Manager to transmit the keystroke, and the Terminal Manager figures out exactly how to transmit the keystroke for a given terminal emulation. Multiple instances of the same tool allows for the same tool to be used by different processes at the same time, like in a MultiFinder environment, or by different threads in a given application. The terminal record is described in greater detail later in this chapter.

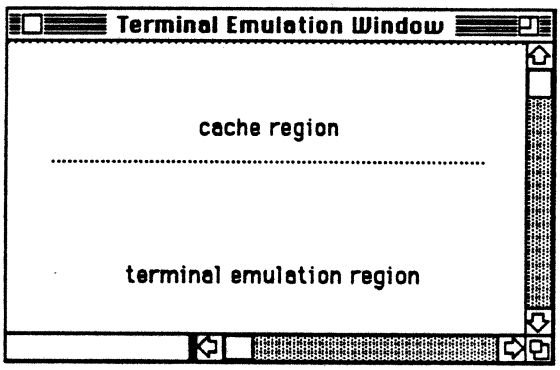
Besides providing basic terminal emulation routines, the Terminal Manager includes routines that make it easy for applications to configure a terminal tool, either through presenting the user with a dialog box or by interfacing directly with a scripting language. The Terminal Manager also contains routines that make it easier for you to localize your applications into foreign languages.

You can use the Terminal Manager in conjunction with other parts of Special K to create a communications application with basic connection, terminal emulation, and file transfer capabilities. Or, you can use the Termianl Manager from Special K, but substitute some other connection service or file transfer service in place of Special K's Connection Manager and File Transfer Manager. You can also write your own terminal tool and add it to the Terminal Manager. (This procedure is discussed in Chapter 8, "Fundamentals of Writing Your Own Tool.") Regardless of which you choose, your application should be able to handle different terminal tools such that users can change tools and still be able to use your program.

The terminal emulation window

The Terminal Manager provides terminal tools with a terminal emulation window. Other than the standard user interface elements, there are two major parts to the terminal emulation window: the terminal emulation region and the cache region. Figure 4-2 shows the major parts of a terminal emulation window.

- **Figure 4-2** Major parts of a terminal emulation window.



The terminal emulation region

The terminal emulation region is the area of the terminal window in which the terminal tool displays data in a manner that emulates a specific terminal. Each line in the terminal emulation region has a corresponding record in the *terminal emulation buffer*, which is stored in a `TermDataBlock` data structure that supports both styled text and graphics information. The format of `TermDataBlock` is:

```
TermDataH           =     ^TermDataPtr;
TermDataPtr         =     ^TermDataBlock;
TermDataBlock       =     RECORD
      flags         :     INTEGER;      {type of block}
      theData       :     Handle;       {text or picts}
      auxData       :     Handle;       {any styled info}
      reserved      :     LONGINT;      {general fudge factor}
```

`theData` is a handle that references the text.

The cache region

The cache region is the area of the terminal window in which your application can display lines of data that would have otherwise scrolled off of the top of the terminal emulation region. Your application does not need to provide a cache region if you don't want it to. But because terminal tools do not support this area of the terminal emulation window, your application must provide all the necessary code if you want a cache region. Your application can take advantage of Terminal Manager routines to implement this feature.

The terminal record

The terminal record contains both information that describes a terminal emulation, as well as pointers to Terminal Manager internal data structures. The Terminal Manager uses this information to "translate" the protocol-independent routines used by an application or tool into a service implemented according to a specified terminal emulation. Most of the fields in the File Transfer record are filled in when an application calls `TMNew`, which is described later in this chapter.

Because the context for a given terminal emulation is maintained in a terminal record, an application can maintain more than one terminal emulation at the same time. All the application has to do is create a new terminal record every time it initiates a terminal emulation.

△ **Important** Your application, in order to be compatible with future releases of the Terminal Manager, should not directly manipulate the fields of the terminal record. The Terminal Manager provides routines that applications and tools can use to change terminal record fields. △

The terminal record data structure

```
TYPE
      TermHandle          =     ^TermPointer;
      TermPointer         =     ^TermRecord;
      TermRecord          =     RECORD
            procID        =     INTEGER


            flags         :     LONGINT;
            errCode       :     TMErr;

            refCon        :     LONGINT;
            userData      :     LONGINT;

            defProc       :     ProcPtr;

            config        :     Prt;
            oldConfig           :     Ptr;

            environsProc  :     ProcPtr;
            reserved1     :     LONGINT;
            reserved2     :     LONGINT;

            private             :     Ptr;

            sendProc      :     ProcPtr;
            breakProc     :     ProcPtr;
            cacheProc     :     ProcPtr;
            clikLoop      :     ProcPtr;


            owner         :     WindowPtr;
            termRect      :     Rect;
            viewRect      :     Rect;
            visRect       :     Rect;

            lastIdle      :     LONGINT;

            selection     :     TMSelection;
            selType       :     INTEGER;

            mluField      :     SearchBlockPtr;
      END;
```

procID

`procID` is the terminal tool ID. This value is dynamically assigned by the Terminal Manager.

flags

flags is a bit field with the following masks:

```
CONST
      tmInvisible          =      1;
      tmSaveBeforeClear    =      2;
      tmNoMenus            =      4;
      tmSaveBeforeDV       =      8;
```

If your application sets TMInvisible, the Terminal Manager will not display the terminal emulation. Instead, it will maintain a virtual terminal emulation; the application can use this virtual terminal emulation to create some other presentation service.

If your application sets TMSaveBeforeClear, the terminal tool will try to cache the entire terminal emulation region in response to any clear screen operation. Clear screen operations are generated from either a user's request, a clear screen character sequence, or a terminal reset character sequence.

If your application sets TMNoMenus, the terminal tool will not put up any custom menus.

If your application sets TMAutoScroll, the terminal tool will automatically scroll the terminal emulation window (if needed) while the user is highlighting a selection.

errCode

errCode is not used by the Terminal Manager in this release.

refCon

refCon is a LONGINT that the application uses to distinguish one terminal record from another in a MultiFinder environment.

userData

userData is a four-byte field that the application can use to store and access values for any purpose.

defProc

defProc is a pointer to the main code resource of the terminal tool that will implement the specifics of the terminal emulation.. The terminal tool's main code resource is of type cdef.

config

config is a pointer to a data block that is private to the terminal tool. The terminal tool uses this record to store terminal information. You can find a description of config later in this manual in Chapter 8, "Fundamentals of Writing Your Own Tool." However, as an application developer, you don't need to be very concerned with this field. All you need to know is that the terminal tool, when selected, will fill in config. To see how this is done, read "Configuring a Terminal Tool" on page nn.

oldConfig

oldConfig is a pointer to a data block that is private to the terminal tool and contains an "old" version of config. This data block is used to implement "undo" operations.

environsProc

environsProc is a pointer to a routine in your application that the terminal tool can call to obtain a record describing the connection environment.

reserved1 and reserved2

reserved1 and reserved2 are reserved for the Terminal Manager.

private

private is a pointer to a data block that is private to the terminal tool. Your application should not use this field.

sendProc

sendProc is a pointer to a routine that your application will call when it needs to send data to another application. A more detailed description of sendProc is later in this chapter under the heading "Routines that must be in your application" on page nn.

breakProc

breakProc is a pointer to a routine in the application that performs a break operation. The effect the break has depends on the terminal emulation being used. A more detailed description of breakProc is later in this chapter under the heading "Routines that must be in your application" on page nn.

cacheProc

cacheProc is a pointer to a routine in the application that is used to save lines that scroll off the top of the terminal emulation region. This routine is also used to save the terminal screen before a clear screen operation (if the TMSaveBeforeClear bit is set in the flags field of the terminal record). A more detailed description of cacheProc is later in this chapter under the heading "Routines that must be in your application" on page nn.

clikLoop

clikLoop is a pointer to a routine that is in the application that handles mouse clicks. The terminal tool calls the click loop repeatedly during click and drag operations. A more detailed description of clickLoop is later in this chapter under the heading "Routines that must be in your application" on page nn.

owner

owner is a pointer to the grafPort where the terminal emulation is performed.

termRect

termRect is the portRect of the current window, which represents the boundaries of the terminal emulation region. Figure 4-3 shows how termRect relates to the terminal emulation window.

viewRect

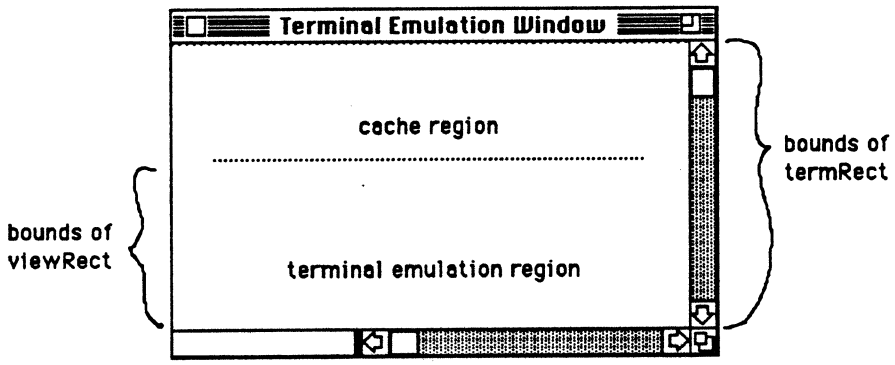
viewRect is a subset of the termRect, which represents the boundaries of the visible part of the terminal emulation region. Figure 4-3 shows how viewRect relates to the terminal emulation window.

visRect

visRect is a rectangle that represents the currently visible rows and columns in the terminal emulation region (for text terminals). It is the same dimensions as viewRect, but is defined by coordinates that represent rows and column numbers. Numbering of rows and columns begins with the number 1.

visRect.top and visRect.left is the top-most line and left-most column that is visible in the terminal emulation region. visRect.bottom and visRect.right is the bottom-most line and right-most column that is visible in the terminal region. These are used by the application to determine scroll-bar values.

■ **Figure 4-3** Bounds of viewRect and termRect.



lastIdle

lastIdle is the last time (in ticks) that the idle procedure was called for the specified terminal record.

selection

`selection` is a data structure that describes the extent of the current selection in the terminal emulation window. Since `selection` can describe either a rectangle or a region, it describes the selection in one of two kinds of data structures: a `Rect` or a `RgnHandle`.

```
TYPE
      TMSelection                =      RECORD
            CASE INTEGER OF
            1:      (
                    selRect      :      Rect;
                    );
            2:      (
                    selRgnHandle :      RgnHandle;
                    );
            END;
```

`selRect` is of type `Rect` and describes the rectangle that has been selected. On text terminals it contains the row/column pairs, with counting beginning at 1. For graphics terminals, it contains pixel coordinates, with (1,1) being the `topLeft` of the terminal region.

If the terminal is a graphics terminal and the selection is a MacPaint-style lasso, selection is a `selRgnHandle` that represents the selection region.

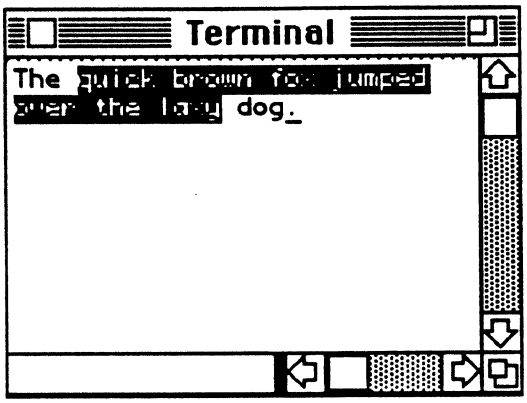
`selType` is a field that further describes a selection; it indicates the highlighting mode that is used to show the selection. Valid values are:

```
CONST
      selTextNormal              =      1;
      selTextBoxed               =      2;
      selGraphicsMarquee         =      4;
      selGraphicsLasso           =      8;
```

Figure 4-4 and 4-5 show that even though two selections may have the same coordinates, different values for `selType` yield different highlighting results.

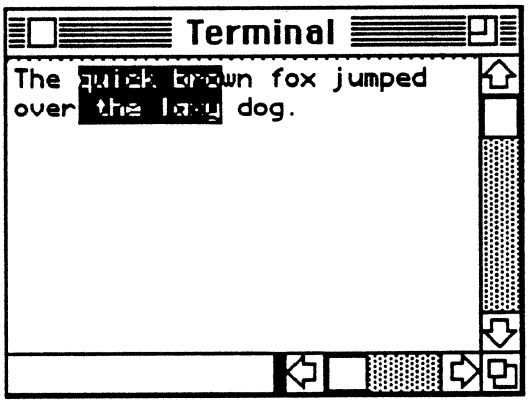
`selTextNormal` is the text selection mode shown in Figure 4-4.

- **Figure 4-4** `selTextNormal` text selection



`selTextBoxed` is also a text selection, but yields different highlighting results as shown if Figure 4-5.

- **Figure 4-5** `selTextBoxed` text selection



`selGraphicsMarquee` is a standard MacPaint-style marquee. `selGraphicsLasso` is a standard MacPaint-style lasso.

`mluField`

`mluField` is a pointer to a linked list of `searchBlock` data structures. The fields in a `searchBlock` are shown in "Searching the terminal emulation buffer" on page nn.

Terminal Manager routines

This sections describes the routines that applications use to access Terminal Manager services.

```
InitTM / nn                    TMGetProcID / nn
TMNew / nn                     TMDefault / nn
TMValidate / nn                TMChoose / nn
TMSetupPreFlight / nn          TMSetupSetup / nn
TMSetupFilter / nn             TMSetupItem / nn
TMSetupCleanup / nn            TMSetupPostflight / nn
TMGetConfig / nn               TMSetConfig / nn
TMStream / nn                  TMPaint / nn
TMIdle / nn                    TMAddSearch / nn
TMRemoveSearch / nn            TMClearSearch / nn
TMGetLine / nn                 TMScroll / nn
TMClear / nn                   TMReset / nn
TMResize / nn                  TMDispose / nn
TMSetSelection / nn            TMGetSelect / nn
TMMenu / nn                    TMActivate / nn
TMResume / nn                  TMClick / nn
TMKey / nn                     TMUpdate / nn
TMEvent / nn                   TMIntlToEnglish / nn
TMEnglishToIntl / nn           TMGetTermEnvirons / nn
TMGetTermName / nn             TMSetRefCon / nn
TMGetRefCon / nn               TMSetUserData / nn
TMGetUserData / nn             TMGetVersion / nn
TMGetTMVersion / nn            TMGetCursor / nn
TMDoTermKey / nn               TMCountTermKeys / nn
TMGetIndTermKey / nn           MySendProc / nn
MyBreak / nn                   Sending a break / nn
MyCache / nn                   SearchCallBack / nn
TMClick / nn                   Click looping / nn
MyGetConnEnvirons / nn
```

Preparing for a terminal emulation

Before your application can start a terminal emulation, it must first initialize the Terminal Manager (`InitTM`), find out the `procID` of the tool it requires (`TMGetProcID`), create a terminal record (`TMNew`), and then configure the terminal tool (`TMChoose`).

`InitTM` Initializing the Terminal Manager

`InitTM` initializes the Terminal Manager. Your application should call this routine after calling the standard Macintosh toolbox initialization routines. If your application uses either the Communications Resource Manager or the Special K Utilities, it should initialize them before initializing the Terminal Manager.

Function

```
InitTM:  TMErr;
```

Description `InitTM` will return an operating system error code if appropriate. If no tools are installed in the Terminal Manager, it will return `tmNoTools`. Your application is responsible to check for the presence of the Communications Toolbox before calling this function.

`TMGetProcID` Getting current procID information

Your application should call `TMGetProcID` just before creating a new terminal record to find out the `procID` of a tool.

Function

```
TMGetProcID(name: STR255): INTEGER;
```

Description `name` specifies a terminal tool. If a terminal tool exists with the specified `name`, its tool ID is returned. If `name` references a nonexistent terminal tool, -1 is returned.

`TMNew` Creating a terminal record

Once the Terminal Manager has been initialized, your application needs to create a terminal record to describe the terminal emulation that is to take place. `TMNew` creates a new terminal record , fills in the fields that it can based on the parameters that were passed to it, and returns a handle to the new record in `TermHandle`. The Terminal Manager then loads the terminal tool's main definition procedure, moves it high in the application heap, and locks it. If memory constraints prevent a new connection record from being created, `TMNew` passes back `NIL` in `TermHandle`.

Function

```
TMNew(termRect: Rect; viewRect: Rect; flags: LONGINT;
procID: INTEGER; owner: WindowPtr; sendProc: ProcPtr;
cacheProc: ProcPtr; breakProc: ProcPtr; clikLoop: ProcPtr;
environsProc: ProcPtr; refCon: LONGINT; userData:
LONGINT): TermHandle;
```

Description

`termRect` is a rectangle in local coordinates that represents the boundaries of the terminal emulation region. Your application initially sets this value by passing it as a parameter to `TMNew`, but the terminal tool may resize it.

`viewRect` is a subset of the `termRect`, which the terminal tool can actually write into. Your application initially sets this value by passing it as a parameter to `TMNew`, but the terminal tool may resize it.

`flags` is a bit field with the following masks:

```
CONST
    tmInvisible             =     1;
    tmSaveBeforeClear       =     2;
    tmNoMenus               =     4;
```

If `tmInvisible` is set, the Terminal Manager will not display the terminal emulation. Instead, it will maintain a virtual terminal emulation; the application can use this virtual terminal emulation to create some other presentation service.

If `tmSaveBeforeClear` is set, the terminal tool will try to cache the entire terminal emulation region in response to any clear screen operation. Clear screen operations are generated from either a user's request, a clear screen character sequence, or a terminal reset character sequence.

If `tmNoMenus` is set, the terminal tool will not put up any custom menus.

`procIDs` are dynamically assigned by the Terminal Manager to tools at run time. Applications should not store `procIDs` in "settings" files. Instead, they should store tool names, which can be converted to `procIDs` with `TMGetProcID`. Use the ID that `TMGetProcID` returns for `procID`.

`owner` is a pointer to the window in which your application is displaying the terminal emulation. If `tmInvisible` is `FALSE`, `owner` should be a `grafPort` that the terminal tool has control over.

`sendProc` is a pointer to a routine that your application will call when it needs to send data on a connection. A more detailed description of `sendProc` is later in this chapter under the heading "Routines that must be in your application" on page nn.

`cacheProc` is a pointer to a routine that is in your application that is used to save lines that scroll off the top of the terminal emulation region. This routine is also used to save the terminal screen before a clear screen operation (if `TMSaveBeforeClear` is set). A more detailed description of `cacheProc` is later in this chapter under the heading "Routines that must be in your application" on page nn.

`breakProc` is a pointer to a routine in your application that performs some sort of break operation. The effect the break has depends upon the terminal emulation tool that your application is using. A more detailed description of `breakProc` is later in this chapter under the heading "Routines that must be in your application" on page nn.

`clikLoop` is a pointer to a routine in your application that is called when the mouse button is held down. The terminal tool calls the click loop repeatedly during click and drag operations. A more detailed description of `clickLoop` is later in this chapter under the heading "Routines that must be in your application" on page nn.

`environsProc` is a pointer to a routine that your application can call when it wants to get information about the connection. Read Chapter 3, "Connection Manager" to find out about `environsProc`.

`userData` and `refCon` are fields for use by the application. `refCon` takes on special meaning in a multiple-connection environment and is used to distinguish one connection record from another.

`TMDefault` Initializing the terminal record

`TMDefault` fills the configuration record pointed to by `theConfig` with the default configuration, which is specified by the terminal tool with the given `procID`. This procedure is called automatically by `TMNew` when filling in fields in a new terminal record.

Procedure

```
TMDefault(VAR theConfig: Ptr; procID: INTEGER; allocate:
BOOLEAN);
```

Description

If `allocate` is `TRUE`, the tool allocates space for `theConfig` in the current zone.

`TMValidate` Validating the terminal record

`TMValidate` validates the current configuration and private data records of the terminal record by comparing the fields in the terminal record with the values that are specified in the terminal tool. This routine is called by `TMNew` and `TMSetConfig` after they have created a new terminal record to make sure that the the record contains values with those specified by the terminal tool.

Function

```
TMValidate(hTerm: TermHandle): BOOLEAN;
```

Description

If the validation fails, `FALSE` is returned and the configuration record is filled with default values for the specified terminal tool.

TMChoose — Configuring a terminal tool

An application can select a terminal tool three ways. The easiest and most straightforward way is by calling the `TMChoose` routine. This routine presents the user with a dialog box similar to the one shown in figure 4-6. The second way that an application can select a terminal tool is by presenting the user with a custom dialog box. This method is much more difficult and involves calling six routines. The routines are described later in this section under "Custom configuration of a terminal tool" and some example code is provided in Appendix B, "Useful code samples" to help you implement this functionality. The third way that your application can select a terminal tool is by interfacing directly with a scripting language. This method allows your application to bypass user interface elements.

To present the user with the standard tool-selection dialog box, your application needs to call `TMChoose`.

Function

```
TMChoose(VAR hTerm: TermHandle; where: Point; idleProc:
ProcPtr): INTEGER;
```

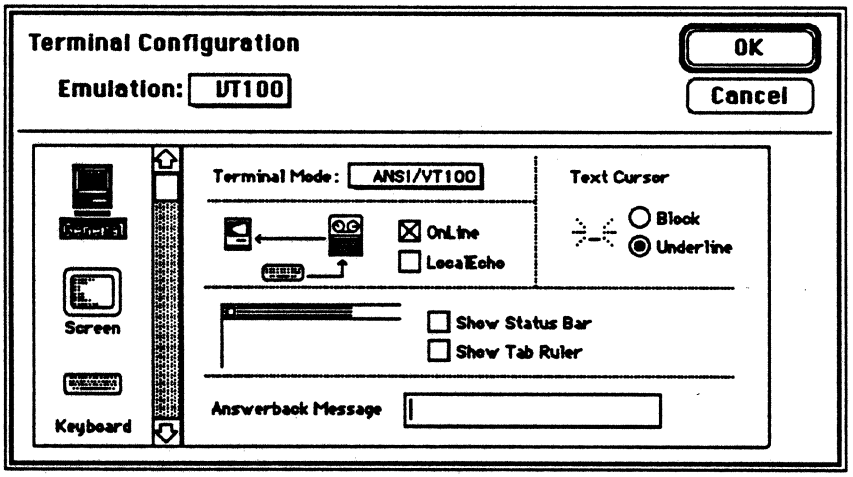
Description

`where` is a point in global coordinates specifying the top left corner of where the dialog box should appear. It is recommended that your application place the dialog box as close to the top and left of the screen as possible.

`idleProc` is a procedure that the Terminal Manager will automatically call every time your application loops through the setup dialog box filter procedure.

`TMChoose` will present the user with a dialog box that looks similar to Figure 4-6

- Figure 4-6 The Standard tool-selection dialog box



`TMChoose` will return one of the following values:

```
CONST
        chooseDisaster      =       -2;
        chooseFailed        =       -1;
        chooseAborted       =       0;
        chooseOKMinor       =       1;
        chooseOKMajor       =       2;
        chooseCancel        =       3;
```

`chooseDisaster` means that the choose operation failed and destroyed the connection record.

`chooseFailed` means that the choose operation failed and the connection record was not changed.

`chooseAborted` means that the user tried to change the connection while it was still open, thereby failing to complete the `CMChoose` operation.

`chooseOKMinor` means that the user selected OK in the dialog box, but did not change the connection tool being used.

`chooseOKMajor` means that the user selected OK in the dialog box and also changed the connection tool being used. The Connection Manager destroys the old connection handle by calling `CMDispose` (the connection is closed down and all pending reads and writes are terminated) and a new connection handle is returned in `hConn`.

`chooseCancel` means that the user selected Cancel in the dialog box.

Custom configuration of a terminal tool

To present the user with a custom tool-configuration dialog box, your application needs to call a series of six Terminal Manger routines: `TMSetupPreflight`, `TMSetupSetup`, `TMSetupItem`, `TMSetupFilter`, `TMSetupCleanup`, and `TMSetupPostflight`. Needless to say, using these routines is a bit more involved than calling `TMChoose`, but they provide your application with much more flexibility. There is a code sample in Appendix B, "Useful code samples" that shows how an application calls these routines.

To build a list of available terminal tools, use the routine `CRMGetIndToolName`, which is described in Chapter 6, "Communications Resource Manager."

TMSetupPreflight

TMSetupPreflight returns a handle to a dialog item list from the terminal tool that the your application should append to the configuration dialog box. (Your application can use AppendDITL which is discussed in Chapter 7, "Special K Utilities.") TMSetupPreflight returns a value in magicCookie that should be passed into the other procedures.

The terminal tool can use TMSetupPreflight to allocate a block of private storage, and to store the pointer to that block in magicCookie. This value, magicCookie, should be passed to the other routines that are used to setup the configuration dialog box.

Function

```
TMSetupPreflight(procID: INTEGER; VAR magicCookie:
LONGINT): Handle;
```

Description

procID is the ID for the terminal tool that is being configured. Your application should get this value by using the TMGetProcID routine, which is discussed later in this chapter.

The refcon of the custom dialog box should point to the procID of the tool being configured.

TMSetupSetup

TMSetupSetup tells the terminal tool to set up controls (like radio buttons or check boxes) in the dialog item list returned by TMSetupPreflight .

Procedure

```
TMSetupSetup(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR magicCookie: LONGINT);
```

Description

procID is the ID for the terminal tool that is being configured. Your application should get this value by using the TMGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

magicCookie is the value returned from the TMSetupPreflight.

TMSetupFilter

TMSetupFilter should be called as a filter procedure prior to the standard modal dialog box filter procedure for the configuration dialog box. This routine allows terminal tools to filter events in the configuration dialog box.

Function

```
TMSetupFilter(procID: INTEGER; theConfig: Ptr;
count:INTEGER; theDialog: DialogPtr; VAR theEvent:
EventRecord; VAR theItem: INTEGER; VAR magicCookie:
LONGINT): BOOLEAN;
```

Description

procID is the ID for the terminal tool that is being configured. Your application should get this value by using the TMGetProcID routine, which is discussed later in this chapter.

count is the number of the first item in the dialog item list appended to the dialog box.

theConfig is the pointer to the configuration record for the tool being configured.

theDialog is the dialog box performing the configuration.

theEvent is the event record for which filtering is to take place.

theItem can return the appropriate item clicked on in the dialog box.

magicCookie is the value returned from TMSetupPreflight.

If the event passed in was handled, TRUE is returned. Otherwise, FALSE indicates that standard dialog box filtering should take place.

TMSetupItem

TMSetupItem processes mouse events for a control in the custom configuration dialog box.

Procedure

```
TMSetupItem(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR item: INTEGER; VAR
magicCookie: LONGINT);
```

Description

procID is the ID for the terminal tool being configured. Your application should get this value by using the TMGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

item is the item clicked on in the dialog box. This value can be modified and sent back.

magicCookie is the value returned from TMSetupPreflight.

TMSetupCleanup

TMSetupCleanup disposes of any storage allocated in TMSetupPreflight and performs any other clean-up operation.

Procedure

```
TMSetupCleanup(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR magicCookie: LONGINT);
```

Description

procID is the ID for the terminal tool that is being configured. Your application should get this value by using the TMGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

magicCookie is the value returned from TMSetupPreflight.

TMSetupPostflight

TMSetupPostflight either shortens or disposes of the dialog box. It will close the tool file if it is not being used by any other sessions.

Procedure

```
TMSetupPostflight(procID:INTEGER);
```

Description

procID is the ID for the terminal tool that is being configured. Your application should get this value by using the TMGetProcID routine, which is discussed later in this chapter.

Scripting language interface

The two routines described below make it easier for your application to configure a terminal record by interfacing with a scripting language, thus bypassing the user interface dialog boxes.

TMGetConfig

`TMGetConfig` returns a null-terminated string from the terminal tool (an example of which is shown after the description of the next routine) containing tokens that fully describe the configuration of the terminal record.

Function

```
TMGetConfig(hTerm: TermHandle): INTEGER;
```

Description

If an error occurs, `TMGetConfig` will return `NIL`. It is the responsibility of your application to dispose of `Ptr`.

TMSetConfig

`TMSetConfig` passes a null-terminated string to the terminal tool for parsing (an example of which is shown under "A sample null-terminated configuration string") that is pointed to by `thePtr`. The string, which can be any length, must contain tokens that describe the configuration of the terminal record, and is parsed from left to right.

Function

```
TMSetConfig(hTerm: TermHandle; thePtr: Ptr): INTEGER;
```

Description

Items that are not recognized or relevant are ignored; this causes `TMSetConfig` to abort parsing the string and to return the character position where the error occurred. If parsing is successfully completed, `TMSetConfig` will return `tmNoErr`. `TMSetConfig` may also return -1 to indicate a general problem with processing the configuration string..

The parsing operation is the responsibility of the individual tool.

A sample null-terminated configuration string

```
FontSize 9 Width 80 Cursor Underline Online True
LocalEcho False AutoRepeat True RepeatControls False
AutoWrap False NewLine False SmoothScroll False
Transparant False SwapBSDelete False \0
```

Using terminal emulation routines

Once your application has performed the required tasks described above, it can then use the routines described next to perform terminal emulations.

TMStream — Putting data into the terminal emulation buffer

Your application should use TMStream to tell the terminal tool where to find data to put into its terminal emulation buffer.

Function

```
TMStream(hTerm: TermHandle; theBuffer: Ptr; length:
LONGINT): LONGINT;
```

Description

theBuffer is the data that is to be placed in the terminal emulation buffer. Typically the data that is pointed to by theBuffer has been provided by the connection tool that your application is using.

TMStream returns the number of bytes that it processed.

TMPaint — Drawing into the terminal emulation region

The TMPaint draws the data in theTermData into the rectangle theRect, which is in local window coordinates.

Procedure

```
TMPaint(hTerm: TermHandle; theTermData:TermDataBlock;
theRect: Rect);
```

Description

theTermData.theData must be a handle to a block on the heap.

TMIdle — Providing necessary idle time

Your application should call TMIdle at least once every time it goes through its main event loop so that the terminal tool can perform idle loop tasks (like blinking the cursor).

Procedure

```
TMIdle(hTerm: TermHandle);
```

TMGetLine — Getting lines from the terminal emulation buffer

TMGetLine returns a line from the terminal emulation buffer.

Procedure

```
TMGetLine(hTerm: TermHandle; lineNo: INTEGER; VAR
theTermData:TermDataBlock);
```

Description

lineNo specifies the line number of a line of data in the terminal emulation buffer(line numbering in the buffer begins with the first line being numbered 1). theTermData contains line data,character attributes, and line attributes.

Your application must allocate `theTermData.theData` with a length of 0 (`theTermData.theData=NewHandle(0)`). The terminal tool copies the information it needs, and increases the size of the handle if it needs to.

`TMScroll` Scrolling the terminal emulation region

`TMScroll` causes the terminal emulation region to scroll either horizontally, vertically, or both.

Procedure

```
TMScroll(hTerm: TermHandle; dH, dV: INTEGER);
```

Description

dH and dV specify the number of pixels to scroll horizontally and vertically. By specifying positive values for dH and dV, the terminal emulation region will scroll down and to the right. By specifying negative values, the terminal emulation region will scroll up and to the left.

`TMClear` Clearing the terminal emulation region

`TMClear` causes the terminal to clear the display screen and to place the default cursor in the home position. Nothing is transmitted to the remote computer.

Procedure

```
TMClear(hTerm: TermHandle);
```

`TMReset` Resetting the terminal

When your application calls `TMReset`, the terminal tool puts the specified terminal into a state that makes it appear as if the terminal had just been turned on. In actuality, the screen representation structure and internal state tables (if the tool has one) are reset to the values specified by the terminal tool, and the configuration record for the terminal is reset to its last saved state.

Procedure

```
TMReset(hTerm: TermHandle);
```

`TMResize` Resizing the terminal region

`TMResize` resizes the terminal emulation region to the coordinates specified in `newViewRect`.

Procedure `TMResize(hTerm: TermHandle; newViewRect: Rect);`

`TMDispose` Disposing of a terminal record

`TMDispose` disposes of the terminal record and all associated data structures and controls.

Procedure `TMDispose(hTerm: TermHandle);`

△ **Important** Your application must call `TMDispose` before disposing of the owning window with `DisposeWindow`. Since `DisposeWindow` clears all controls in the control list, a subsequent call to `TMDispose` may cause problems.△

Searching the terminal emulation buffer

Searching the terminal emulation buffer can take place anytime that an application or tool wants it to, but typically a tool will perform a search during its idle procedure. To tell a tool to search for a specified string, your application needs to call the `TMAddSearch` routine. To tell the terminal tool to stop performing a search, your application must use `TMRemoveSearch`. To tell the terminal tool to stop all searches, your application must use `TMClearSearch`.

`TMAddSearch`

Function
```
TMAddSearch(hTerm: TermHandle; theString: STR255; where:
Rect; searchType: INTEGER; callBack: ProcPtr): INTEGER;
```

Description This function returns a reference number that is assigned to each search, or -1 if the search was not successfully added. The tool will search for `theString` in the area specified by `where`. `where` is a rectangle that contains two row/column pairs, with numbering of rows and columns starting with the number 0.

By specifying a -1 as a value in the row/column pairs, your application can limit the search to one row, one column, or the intersection of one row and one column. The next table shows how your application can use -1as a search region delimiter.

To Search In:	Use Row/Column Pair:

rectangle bounded by n,m,o,p	(n,m) (o,p)
row n	(n, -1) (-1, -1)
column m	(-1, m) (-1,-1)
rows n-o (inclusive)	(n, -1) (o, -1)
column m-p (inclusive)	(-1, m) (-1, p)
anywhere	(-1, -1) (-1, -1)

`searchType` is one of the two below, examples of which can be seen in Figures 4-4 and 4-5.

```
CONST
selTextNormal   =   1;
selTextBoxed    =  2;
```

`callBack` is a procedure the tool will automatically call when it finds a match. `callBack` must be supplied by your application, and is described later in this chapter under "Routines that must be in your application" on page nn.

TMRemoveSearch

`TMRemoveSearch` stops the search specified by `refNum`.

Procedure

```
TMRemoveSearch(hTerm: TermHandle; refNum: INTEGER);
```

TMClearSearch

`TMClearSearch` stops all searches for the terminal record.

Procedure

```
TMClearSearch(hTerm: TermHandle);
```

Manipulating selections

The terminal manager provides two routines that make it easier for your application to manipulate selections. `TMSetSelection` highlights a selection in the terminal emulation window and `TMGetSelect` goes out and gets the selection.

TMSetSelection

`TMSetSelection` sets the current selection to be defined as `theSelection`.

Procedure

```
TMSetSelection(hTerm: TermHandle; theSelection:
TMSelection; selType: INTEGER);
```

Description

`selType` determines the type of highlighting for the selection (examples of which can be seen in Figures 4-4 and 4-5) and may be:

```
CONST
    selTextNormal               =       1;
    selTextBoxed                =       2;
    selGraphicsMarquee          =       4;
    selGraphicsLasso            =       8;
```

TMGetSelect

`TMGetSelect` returns either the number of bytes in the selection in the terminal emulation window or an appropriate operating system error code.

Function

```
TMGetSelect(hTerm: TermHandle; theData: Handle; VAR
theType: ResType): LONGINT;
```

Description

`theData` must be a handle to a block of size 0. `TMGetSelect` will resize this block as necessary.

`theType` returns the type of data selected. If there is no selection that is active, `TMGetSelect` returns 0.

Handling events

The Terminal Manager event processing routines provide useful extensions to the Macintosh Toolbox Event Manager. The section below explains the six routines that Special K provides. There is example code in Appendix B, "Useful code samples" that shows how an application can determine if an event needs to be handled by one of these procedures.

TMMenu — Handling menu events

Your application must call `TMMenu` when the user has made a selection from a menu that is installed by the terminal tool.

Function `TMMenu(hTerm: TermHandle; menuID: INTEGER; item: INTEGER): BOOLEAN;`

Description `TMMenu` returns `FALSE` if the menu item was not handled by the terminal tool. `TMMenu` returns `TRUE` if the terminal tool did handle the menu item.

`TMActivate` Activate events

`TMActivate` processes an activate or deactivate event for a terminal window. The addition or removal of special menus from the menu bar is an example of an operation that would be performed in response to an activate or deactivate event.

Procedure `TMActivate(hTerm: TermHandle; activate: BOOLEAN);`

Description If `activate` is `TRUE`, an activate event is to be processed. Otherwise, a deactivate event is to be processed.

`TMResume` Resume events

`TMResume` processes a resume or suspend event for a terminal window. Resume and suspend events are processed only if a tool has a custom menu to install or remove from the menu bar. If `resume` is `TRUE`, then a resume event is to be processed.

Procedure `TMResume(hTerm: TermHandle; resume: BOOLEAN);`

`TMClick` Mouse events

`TMClick` processes a `mouseDown` in the terminal emulation region. The routine pointed to by `clikLoop`, which is discussed under "Routines that must be in your application" on page nn, is called repeatedly by `TMClick`

Procedure `TMClick(hTerm: TermHandle; theEvent: EventRecord);`

`TMKey` Keyboard events

`TMKey` processes a `keyDown` or `autoKey` event. The keystroke will typically be translated into a sequence of bytes that are then transmitted by calling your application's `MySendProc`. (`MySendProc` is discussed later in this chapter under "Routines that must be in your application" on page nn.)

Procedure `TMKey(hTerm: TermHandle; theEvent: EventRecord);`

`TMUpdate` Update events

Your application will typically call `TMUpdate` between `BeginUpdate` and `EndUpdate`.

Procedure `TMUpdate(hTerm: TermHandle; visRgn: RgnHandle);`

Description `visRgn` specifies the region to be updated.

`TMEvent` Handling other events

Your application can call `TMEvent` in response to receiving an event for a window that belongs to the Terminal Manager. An example of such an event is when the user clicks on a button in a dialog box that a terminal tool is displaying

Procedure `TMEvent(hTerm: TermHandle; theEvent: EventRecord);`

Description Windows (or dialog boxes) that belong to the Terminal Manager should have a terminal record handle stored in the `refCon` field of the `windowRecord`.

Localizing strings

Special K provides two routines that make it easier to localize strings.

`TMIntlToEnglish`

`TMIntlToEnglish` converts a configuration string that is pointed to by `inputPtr` in the given language to an American English configuration string that is pointed to by `outputPtr`.

Function `TMIntlToEnglish(hTerm: TermHandle; inputPtr: Ptr; VAR outputPtr: Ptr; language: INTEGER): OSErr;`

Description `language` specifies the language from which the string is to be converted.

The terminal tool allocates space for `outputPtr`.

If the language specified is not supported, `tmNoErr` is still returned, but `outputPtr` is `NIL`.

The function returns an operating system error code if any internal errors occur.

TMEnglishToIntl

TMEnglishToIntl converts a configuration string that is pointed to by inputPtr in English to a configuration string in the given language that is pointed to by outputPtr.

Function

```
TMEnglishToIntl(hTerm: TermHandle; inputPtr: Ptr; VAR
outputPtr: Ptr; language: INTEGER): OSErr;
```

Description

language specifies the language to which the string is to be converted.

The terminal tool allocates space for outputPtr.

If the language specified is not supported, tmNoErr is still returned, but outputPtr is NIL.

The function returns an operating system error code if any internal errors occur.

Miscellaneous routines

TMGetTermName

TMGetTermName returns in name the name of the tool specified by id.

Procedure

```
TMGetTermName(id: INTEGER; VAR name: STR255);
```

Description

If id references a terminal tool that does not exist, the Terminal Manager sets name to an empty string.

TMSetRefCon

TMSetRefCon sets the terminal record's reference constant to the specified value.

Procedure `TMSetRefCon(hTerm: TermHandle; rC: LONGINT);`

TMGetRefCon

`TMGetRefCon` returns the terminal record's reference constant.

Function `TMGetRefCon(hTerm: TermHandle): LONGINT;`

TMSetUserData

`TMSetUserData` sets the terminal record's `userData` field to the value specified by `uD`.

Procedure `TMSetUserData(hTerm: TermHandle; uD: LONGINT);`

TMGetUserData

`TMGetUserData` returns the terminal record's `userData` field.

Function `TMGetUserData(hTerm: TermHandle): LONGINT;`

TMGetVersion

`TMGetVersion` returns a handle to a relocatable block that contains the information in the terminal tool's `tvers` resource. The terminal tool must have the same ID as that specified by `hTerm`. This handle is *not* a resource handle.

Function `TMGetVersion(hTerm: TermHandle): Handle;`

TMGetTMVersion

`TMGetTMVersion` returns the version number of the Terminal Manager.

Function `TMGetTMVersion: INTEGER;`

TMGetCursor

`TMGetCursor` returns the current position of the cursor.

Function `TMGetCursor(hTerm: TermHandle; cursType: INTEGER): Point;`

Description Valid values for `cursType` are:

```
CONST
    cursorText          =     1;
    cursorGraphics      =     2;
```

TMDoTermKey

`TMDoTermKey` emulates a special terminal key specified by `theKey`.

Function `TMDoTermKey(hTerm: TermHandle; theKey: STR255): BOOLEAN;`

Description If the key specified by `theKey` is not understood, this routine returns `FALSE`. Otherwise, if the key specified is processed, this routine returns `TRUE`.

The example below shows how an application can use `TMDoTermKey` to emulate the pressing of a PF1 key.

```
IF TMDoTermKey(hTerm, 'PF1') THEN
      BEGIN
      END;
```

TMCountTermKeys

`TMCountTermKeys` returns the number of special terminal keys that the tool supports.

Function `TMCountTermKeys(hTerm): INTEGER;`

TMGetIndTermKey

`TMGetIndTermKey` returns in `theKey` the terminal key specified by `ID`. If `ID` specifies a key that does not exist, this routine returns an empty string.

Procedure

```
TMGetIndTermKey(hTerm:TermHandle; ID:INTEGER; VAR
theKey:STR255);
```

TMGetTermEnvirons Getting general terminal tool information

`TMGetTermEnvirons` returns `theEnvirons`, which reflects the internal conditions of the terminal tool.

Function

```
TMGetTermEnvirons(hTerm: TermHandle; VAR theEnvirons:
TermEnvironRec): TMErr;
```

Description This routine routine will return `tmNoErr`, `envVersTooBig`, or an operating systerm error code. The fields in `theEnvirons` are:

```
TYPE
    TermEnvironPtr          =    ^TermEnvironRec;
    TermEnvironRec          =    RECORD
          version           :    INTEGER;
          termType          :    INTEGER;
          textRows          :    INTEGER;
          textCols          :    INTEGER;
          cellSize          :    Point;
          graphicSize       :    Rect;
          slop              :    Point;
          auxSpace          :    Rect;
    END;
```

`version` is the version of the requested terminal environment record, which is 0 in this release of the Terminal Manager.

`termtype` is the type of terminal, which is one of the following:

```
CONST
    TMTextTerminal          =    1;
    TMGraphicsTerminal      =    2;
```

`textRows` is the number of rows in the terminal emulation region.

`textCols` is the number of columns in the terminal emulation region.

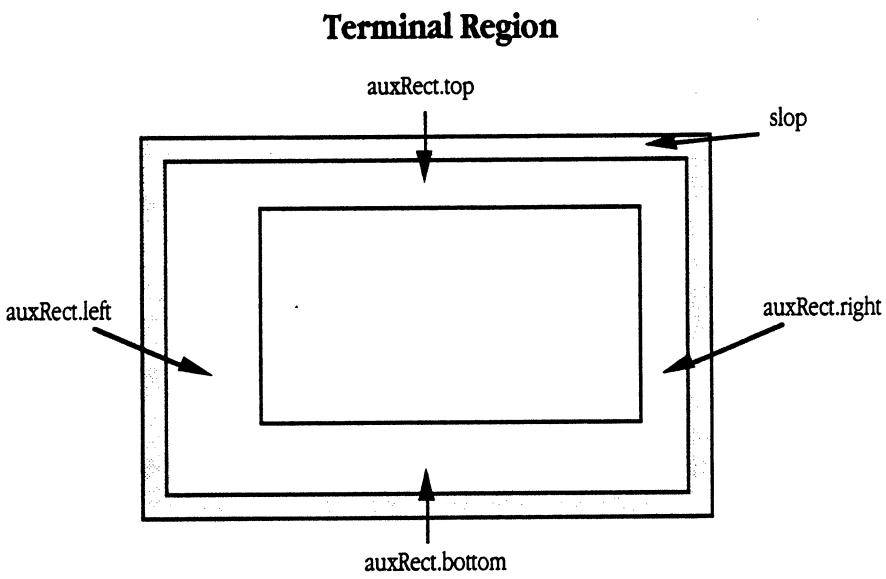
`cellSize` is the height and width of each cell.

`graphicSize` is the size of the graphics terminal tool default rectangle measured in pixels.

`slop` is the border of the terminal region.

`auxSpace` is a rectangle that specifies any additional space that is required at the top, bottom, right, or left of the terminal region, as shown in Figure 4-7

■ **Figure 4-7** Additional space in the terminal emulation region



Routines that must be in your application

Not all of the code necessary to perform a terminal emulation is provided in any of the terminal tools; your application must provide the necessary code (or at least pointers to code provided by other managers). Below are the routines that must be in your application, which tell the tool

- how to send data on the connection
- what to do with lines that scroll out of the terminal emulation region
- what to do when a specified string is found in the terminal emulation buffer
- what to do when the user whats to effect a break on the terminal
- what to do when the user is dragging the mouse in the terminal emulation region
- what the connection environment is like.

`MySendProc` Sending data out along the connection

When a tool needs to send data out to the other application, it will look to your application to find `MySendProc`. `MySendProc` may simply be the routine that the Connection Manager is using to send data (as is the case in the next example), or you can write a send routine of your own.

Function

```
MySendProc (thePtr: Ptr; theSize: LONGINT; refCon:
LONGINT): LONGINT;
```

Description

`thePtr` is a pointer to the data to be sent.

`theSize` is the number of characters to be sent.

`refCon` is the reference constant field of the sending terminal's terminal.

`MySendProc` returns the actual number of characters sent.

Sample routine for sending data

```
FUNCTION SendProc(thePtr : Ptr; theSize : LONGINT; refcon : LONGINT;
flags: INTEGER) : LONGINT;
VAR
      theWindow : WindowPtr;
      pWindow : WindowP;

      theErr : CMErr;

BEGIN
      theWindow := WindowPtr(refcon);
      theConn:= GethConn(theWindow);
      SendProc := 0;
      IF theConn= NIL THEN
            Exit(SendProc);

      IF WindowPeek(theWindow)^.windowKind <> userKind THEN
            Exit(SendProc);

      pWindow := WindowP(GetWRefCon(theWindow));
      theTerm := pWindow^.hTerm;
      theConn := pWindow^.hConn;

      theErr := CMWrite(theConn, thePtr, theSize, CMData, FALSE, NIL,
0, flags);
      SendProc := theSize;
END;
```

MyBreak — Sending a break

Your applications needs to contain information about how to break a connection. While it can contain the code that performs the break operation, your application can also point to a connection tool routine to do it, as does the next sample.

Procedure `MyBreak(duration: LONGINT; refCon: LONGINT);`

Description `duration` is a time value in ticks that specifies how long the break should last.

`refCon` is the reference constant field of the terminal record.

Sample showing how to break a connection

```
PROCEDURE BreakProc(duration: LONGINT; refcon : LONGINT);
VAR
      theWindow : WindowPtr;
      pWindow : WindowP;

      theErr : CMErr;

BEGIN
      theWindow := WindowPtr(refcon);
      theConn:= GethConn(theWindow);
      IF theConn= NIL THEN
            Exit(BreakProc);
      IF WindowPeek(theWindow)^.windowKind <> userKind THEN
            Exit(BreakProc);

      pWindow := WindowP(GetWRefCon(theWindow));
      theConn := pWindow^.hConn;

      CMBreak(theConn, duration, TRUE, @BreakCompletion);
      {asynchornous with break completion routine BreakCompletion}
END;
```

MyCache — Caching lines from the terminal region

Your application can cache lines that scroll up out of the terminal emulation region and, perhaps, display them in the terminal emulation window. If you want your application to do this, you have to provide the code to support this. If you do not want your application to support this, then your application should specify `NIL` for `MyCache` when it calls `TMNew`.

Function

```
MyCache(refCon: LONGINT; theTermData:TermDataBlock):
LONGINT;
```

Description

`refCon` is the reference constant for the terminal record.

`theTermData` is a data structure of type `TermDataBlock`:

```
TermDataH       =     ^TermDataPtr;
TermDataPtr     =     ^TermDataBlock;
TermDataBlock   =     RECORD
    flags       :     INTEGER;     {type of block}
    theData     :     Handle;      {text or picts}
    auxData     :     Handle;      {any styled info}
    reserved    :     LONGINT;     {general fudge factor}
```

`theTerm.theData` must be a handle to a block on the heap. Your application can calculate the size of this heap with `GetHandleSize`. Your application must copy any data it needs (it can use `HandToHand`) because `theTermData` belongs to the terminal tool and may not exist after `MyCache` has completed.

`MyCache` must return `tmNoErr` if no error occurred during processing, otherwise it must return an appropriate error code.

`SearchCallBack` Responding to a matched search parameter

Your application can selectively filter data in the terminal emulation buffer by making use of a search call-back procedure. Since a tool will automatically call `SearchCallBack` when it finds a match to the search string, your application can respond any way that you program it to.

Procedure

```
SearchCallBack(hTerm: TermHandle; refNum: INTEGER;
foundRect: Rect);
```

Description

`refNum` is a reference number associated with a particular search. These values are assigned by the Terminal Manager when a search is added to a terminal record with the `TMAddSearch` routine.

`foundrect` describes in row/column format where the match was found.

`TMClick` Click looping

This routine is called when the user is dragging the mouse in the terminal emulation window. Initially, your application should process a mouse down event by calling `TMClick`, which in turn calls this routine.

Function

```
MyClikLoop(refCon: LONGINT): BOOLEAN;
```

Description

`TRUE` is returned while the mouse is clicked within the terminal region.

MyGetConnEnvirons Getting connection environment information

Your application might need to pass information about the connection environment to the terminal tool. To accomplish this, the terminal tool will call a routine in the application, `MyGetConnEnvirons`.

Function

```
MyGetConnEnvirons(refCon: LONGINT; VAR theEnvirons:
ConnEnvironRec): CMErr;
```

Description

`refCon` is the reference constant for the terminal tool.

`theEnvirons` is a data structure containing the connection environment record. Your application can either construct `theEnvirons` or use the Connection Manager routine `CMGetConnEnvirons`. For more information about `theEnvirons`, read Chapter 3, "Connection Manager."

The next sample shows how `MyGetConnEnvirons` can point to a Connection Manager routine to retrieve information about the connection environment.

Sample terminal environment routine

```
FUNCTION TermGetConnEnvirons(refCon: LONGINT; VAR theEnvirons:
ConnEnvironRec): OSErr;
VAR
      theWindow : WindowPtr;
      pWindow : WindowP;

BEGIN
      theWindow := WindowPtr(refcon);
      theConn:= GethConn(theWindow);

      TermGetConnEnvirons := envNotPresent;
      IF theConn= NIL THEN
            Exit(TermGetConnEnvirons);

      IF WindowPeek(theWindow)^.windowKind <> userKind THEN
            Exit(TermGetConnEnvirons);

      pWindow := WindowP(GetWRefCon(theWindow));
      theConn := pWindow^.hConn;

      TermGetConnEnvirons := CMGetConnEnvirons(theConn, theEnvirons);
END;
```

Summary

Terminal Manager routines	*see page*
`InitTM:OSErr;`	nn
`TMActivate(hTerm: TermHandle; activate: BOOLEAN);`	nn
`TMAddSearch(hTerm: TermHandle; theString: STR255; where: Rect; searchType: INTEGER; callBack: ProcPtr): INTEGER;`	nn
`TMChoose(VAR hTerm: TermHandle; where: Point; idleProc: ProcPtr): INTEGER;`	nn
`TMClear(hTerm: TermHandle);`	nn
`TMClearSearch(hTerm: TermHandle);`	nn
`TMClick(hTerm: TermHandle; theEvent: EventRecord);`	nn
`TMCountTermKeys(hTerm): INTEGER;`	nn
`TMDefault(VAR theConfig: Ptr; procID: INTEGER; allocate: BOOLEAN);`	nn
`TMDispose(hTerm: TermHandle);`	nn
`TMDoTermKey(hTerm: TermHandle; theKey: STR255): BOOLEAN;`	nn
`TMEnglishToIntl(hTerm: TermHandle; inputPtr: Ptr; VAR outputPtr: Ptr; language: INTEGER): INTEGER;`	nn
`TMEvent(hTerm: TermHandle; theEvent: EventRecord);`	nn
`TMGetConfig(hTerm: TermHandle): Ptr;`	nn
`TMGetCursor(hTerm: TermHandle; cursType: INTEGER): Point;`	nn
`TMGetIndTermKey(hTerm:TermHandle; ID:INTEGER; VAR theKey"STR255);`	nn

```
TMGetLine(hTerm: TermHandle; lineNo: INTEGER; VAR                    nn
            theTermData:TermDataBlock);

TMGetProcID(name: STR255): INTEGER;                                  nn

TMGetRefCon(hTerm: TermHandle): LONGINT;                             nn

TMGetSelect(hTerm: TermHandle; theData: Handle; VAR                  nn
            theType: ResType): LONGINT;

TMGetTermEnvirons(hTerm: TermHandle; VAR theEnvirons:                nn
            TermEnvironRec): TMErr;

TMGetTermName(id: INTEGER; VAR name: STR255);                        nn

TMGetTMVersion: INTEGER;                                             nn

TMGetUserData(hTerm: TermHandle): LONGINT;                           nn

TMGetVersion(hTerm: TermHandle): Handle;                             nn

TMIdle(hTerm: TermHandle);                                           nn

TMIntlToEnglish(hTerm: TermHandle; inputPtr: Ptr; VAR                nn
            outputPtr: Ptr; language: INTEGER): INTEGER;

TMKey(hTerm: TermHandle; theEvent: EventRecord);                     nn

TMMenu(hTerm: TermHandle; menuID: INTEGER; item: INTEGER):           nn
            BOOLEAN;
```

```
TMNew(termRect: Rect; viewRect: Rect; flags: LONGINT;          nn
              procID: INTEGER; owner: WindowPtr; sendProc:
              ProcPtr; cacheProc: ProcPtr; breakProc:
              ProcPtr; clikLoop: ProcPtr; environsProc:
              ProcPtr; refCon: LONGINT; userData:
              LONGINT): TermHandle;

TMPaint(hTerm: TermHandle; theTermData:TermDataBlock;          nn
              theRect: Rect);

TMRemoveSearch(hTerm: TermHandle; refNum: INTEGER);            nn

TMReset(hTerm: TermHandle);                                    nn

TMResize(hTerm: TermHandle; newViewRect: Rect);                nn

TMResume(hTerm: TermHandle; resume: BOOLEAN);                  nn

TMScroll(hTerm: TermHandle; dH, dV: INTEGER);                  nn

TMSetConfig(hTerm: TermHandle; thePtr: Ptr): INTEGER;          nn

TMSetRefCon(hTerm: TermHandle; rC: LONGINT);                   nn

TMSetSelection(hTerm: TermHandle; theSelection:                nn
              TMSelection; selType: INTEGER);

TMSetupCleanup(procID: INTEGER; theConfig: Ptr; count:         nn
              INTEGER; theDialog: DialogPtr; VAR
              magicCookie: LONGINT);

TMSetupFilter(procID: INTEGER; theConfig: Ptr; count:          nn
              INTEGER; theDialog: DialogPtr; VAR theEvent:
              EventRecord; VAR theItem: INTEGER; VAR
              magicCookie: LONGINT): BOOLEAN;

TMSetupItem(procID: INTEGER; theConfig: Ptr; count:            nn
              INTEGER; theDialog: DialogPtr; VAR theItem:
              INTEGER; VAR magicCookie: LONGINT);

TMSetupPostflight(procID:INTEGER);                             nn

TMSetupPreflight(procID: INTEGER; VAR magicCookie:             nn
              LONGINT): Handle;

TMSetupSetup(procID: INTEGER; theConfig: Ptr; count:           nn
              INTEGER; theDialog: DialogPtr; VAR
              magicCookie: LONGINT);

TMSetUserData(hTerm: TermHandle; uD: LONGINT);                 nn
```

```
TMStream(hTerm: TermHandle; theBuffer: Ptr; length:          nn
              LONGINT): LONGINT;

TMUpdate(hTerm: TermHandle; visRgn: RgnHandle);              nn
```

```
TMValidate(hTerm: TermHandle): BOOLEAN;                    nn
```

Routines in your application ***n n***

```
MySendProc (thePtr: Ptr; theSize: LONGINT; refCon:         nn
            LONGINT): LONGINT;

MyBreak(duration: LONGINT; refCon: LONGINT);               nn

MyCache(refCon: LONGINT; theTermData:TermDataBlock):       nn
            LONGINT;

SearchCallBack(hTerm: TermHandle; refNum: INTEGER);        nn

MyClickLoop(refCon: LONGINT): BOOLEAN;                     nn

MyGetConnEnvirons(refCon: LONGINT; VAR theEnvirons:        nn
            ConnEnvironRec): CMErr;
```

Terminal Record

```
TermHandle          =     ^TermPointer;
TermPointer         =     ^TermRecord;
TermRecord          =     RECORD
      procID        =     INTEGER


      flags         :     LONGINT;
      errCode       :     TMErr;

      refCon        :     LONGINT;
      userData      :     LONGINT;

      defProc       :     ProcPtr;

      config        :     Prt;
      oldConfig           :     Ptr;

      environsProc  :     ProcPtr;
      reserved1     :     LONGINT;
      reserved2     :     LONGINT;

      private             :     Ptr;

      sendProc      :     ProcPtr;
      breakProc     :     ProcPtr;
      cacheProc     :     ProcPtr;
      clikLoop      :     ProcPtr;
```

```
        owner           :       WindowPtr;
        termRect        :       Rect;
        viewRect        :       Rect;
        visRect         :       Rect;

        lastIdle        :       LONGINT;

        selection       :       TMSelection;
        selType         :       INTEGER;

        mluField        :       SearchBlockPtr;
    END;


TYPE
    TMSelection                         =       RECORD
        CASE INTEGER OF
        1:      (
                selRect             :       Rect;
                );
        2:      (
                selRgnHandle        :       RgnHandle;
                filler              :       LONGINT;
                );
        END;

    searchBlockPtr                      =       ^searchBlock;
    searchBlock                         =       RECORD
        theString                       :       StringHandle;
        where                           :       Rect;
        searchType                      :       INTEGER;
        callBack                        :       ProcPtr;
        refNum                          :       INTEGER;
        next                            :       searchBlockPtr
    END;


TermDataH           =       ^TermDataPtr;
TermDataPtr         =       ^TermDataBlock;
TermDataBlock       =       RECORD
    flags           :       INTEGER;        {type of block}
    theData         :       Handle;         {text or picts}
    auxData         :       Handle;         {any styled info}
    reserved        :       LONGINT;        {general fudge factor}
```

```
TYPE
      TermEnvironPtr            =     ^TermEnvironRec;
      TermEnvironRec            =     RECORD
            version             :     INTEGER;
            termType            :     INTEGER;
            textRows            :     INTEGER;
            textCols            :     INTEGER;
            cellSize            :     Point;
            graphicSize         :     Rect;
            slop                :     Point;
            auxSpace            :     Rect;
      END;

      END;
```

Constants

```
CONST
{ bit masks for flags field of terminal record }
      tmInvisible         =     1;
      tmSaveBeforeClear   =     2;
      tmNoMenus           =     4;
      tmSaveBeforeDV      =     8;


{ values returned from initTM }
      tmNoErr                   =     0;
      tmNoTools                 =     8;

{     selection types }
      selTextNormal             =     1;
      selTextBoxed              =     2;
      selGraphicsMarquee        =     4;
      selGraphicsLasso          =     8;

{ search modifiers  }
      tmSearchNoDiacrit         =     256;
      tmSearchNoCase            =     512;

{ terminal types in TermEnvironRec data structure  }
      tmTextTerminal            =     1;
      tmGraphicsTerminal        =     2;
```

```
{ Choose return values }
      chooseDisaster      =      -2;
      chooseFailed        =      -1;
      chooseAborted       =      0;
      chooseOKMinor       =      1;
      chooseOKMajor       =      2;
      chooseCancel        =      3;
```

Searching

To Search In:	Use Row/Column Pair:
rectangle bounded by n,m,o,p	(n,m) (o,p)
row n	(n, -1) (-1, -1)
column m	(-1, m) (-1,-1)
rows n-o (inclusive)	(n, -1) (o, -1)
column m-p (inclusive)	(-1, m) (-1, p)
anywhere	(-1, -1) (-1, -1)

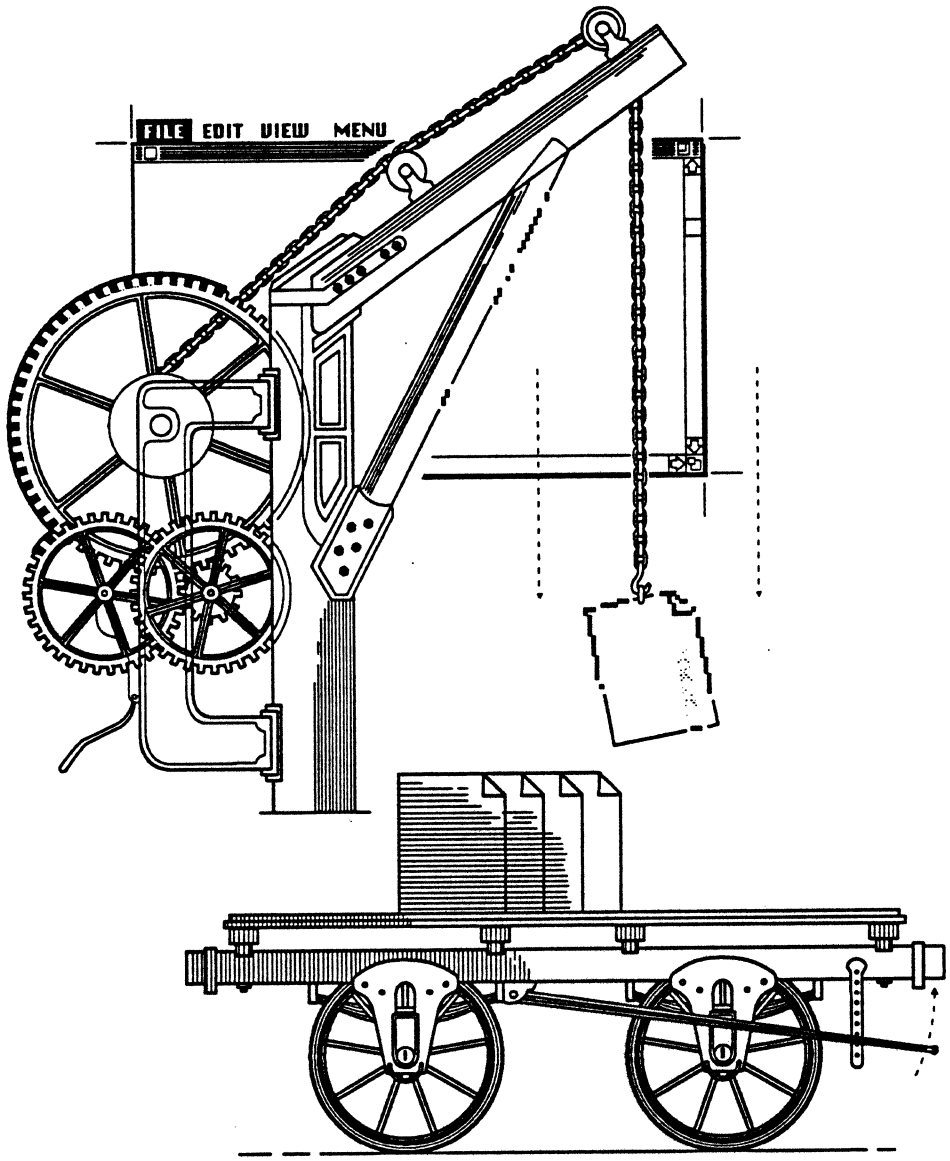
Terminal Manager routine selectors:

```
InitTM            .EQU    769         TMNew             .EQU    770
TMDispose         .EQU    771         TMKey             .EQU    772
TMUpdate          .EQU    773         TMPaint           .EQU    774
TMActivate        .EQU    775         TMResume          .EQU    776
TMClick           .EQU    777         TMStream          .EQU    778
TMMenu            .EQU    779         TMReset           .EQU    780
TMClear           .EQU    781         TMResize          .EQU    782
TMGetSelect       .EQU    783         TMGetLine         .EQU    784
TMSetSelection    .EQU    785         TMScroll          .EQU    786
TMIdle            .EQU    787         TMValidate        .EQU    788
TMDefault         .EQU    789         TMSetupPreflight  .EQU    790
```

```
TMSetupSetup        .EQU    791         TMSetupFilter       .EQU    792
TMSetupItem         .EQU    793         TMSetupCleanup      .EQU    794
TMGetConfig         .EQU    795         TMSetConfig         .EQU    796
TMIntlToEnglish     .EQU    797         TMEnglishToIntl     .EQU    798
TMGetProcID         .EQU    799         TMGetToolName       .EQU    800
TMSetRefCon         .EQU    801         TMGetRefCon         .EQU    802
TMSetUserData       .EQU    803         TMGetUserData       .EQU    804
TMGetVersion        .EQU    805         TMGetTMVersion      .EQU    806
TMAddSearch         .EQU    807         TMRemoveSearch      .EQU    808
TMClearSearch       .EQU    809         TMGetCursor         .EQU    810
TMGetTermEnvirons   .EQU    811         TMChoose            .EQU    812
TMEvent             .EQU    813         TMDoTermKey         .EQU    814
TMCountTermKeys     .EQU    815         TMGetIndTermKey     .EQU    816
TMSetupPostflight   .EQU    817
```

```
TMSetupPostflight .EQU     817
```

Chapter 5 **File Transfer Manager**



About this chapter

This chapter describes the File Transfer Manager, which is the Special K manager that allows applications to implement file transfer services without having to take into account underlying file transfer protocols. This chapter starts out by describing fundamental concepts about the File Transfer Manager. Then it describes the file transfer record, which is the most important record to the File Transfer Manager. After a detailed functional description of routines the Terminal Manager provides, this chapter finishes with a summary you can use as a quick reference to routines and data structures.

Often refered to in this chapter is the term "your application", which is the application you are writing for the Macintosh, and which will implement communication services for users. Be careful not to confuse the services your application is requesting with the services that tools provide.

To use the File Transfer Manager, you need to be familiar with the following topics.

- Resource Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- File Manager (see *Inside Macintosh*, Volumes: II, IV, V)
- Standard File (see *Inside Macintosh*, Volumes: I, IV)
- Connection Manager (see in this manual Chapter 3, "Connection Manager")

About the File Transfer Manager

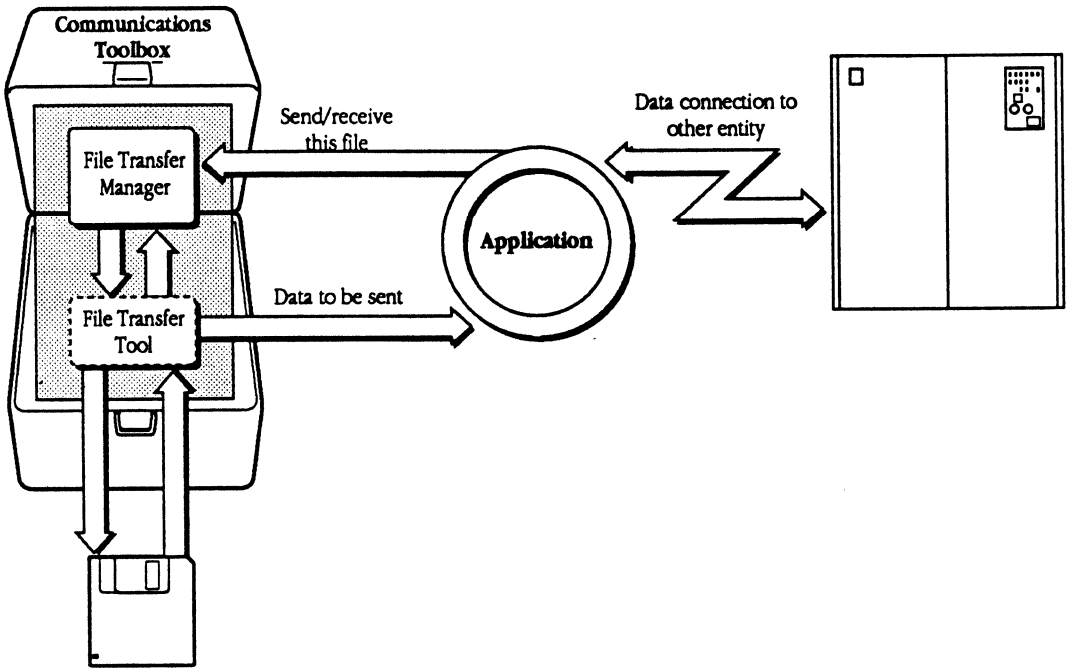
By using File Transfer Manager routines, your application can send files to or receive files from another application without having to take into account underlying file transfer protocols. File transfer tools, which are discussed in Chapter 11, "Writing File Transfer Tools," are responsible for implementing file transfer services according to specific protocols.

To the application, the File Transfer Manager provides a basic abstraction of a file transfer between two entities. These entities could be different processes running on the same CPU or one process running on a Macintosh and the other running on a mainframe (or any other type of computer).

Here's what happens inside the File Transfer Manager. An application makes a request of the File Transfer Manager when it needs it to do something, such as send a file. The File Transfer Manager then sends this request on to one of the tools it manages. The tool, in turn, takes the request and executes the service according to the specifics of the file transfer protocol that is implemented on the data connection. (You can find information about connections in Chapter 3, "Connection Manager") Once the tool has finished, it passes back to the application any relevant parameters and return codes.

Figure 5-1 shows the data flow between into and out of the File Transfer Manager.

Figure 5-1: Data flow into and out of the File Transfer Manager.



The most important data structure maintained by the File Transfer Manager is the *file transfer record*, which is where all the specifics about a file transfer are stored. For example, the file transfer record might show that the File Tranfer Manager should use the XMODEM tool to perform file transers, and that the tool should not display any custom menus while transfering files.

Two reasons why the file transfer record is so important to the File Transfer Manager are that its existence allows for both protocol-independent routines and multiple instances of the same tool. Protocol-independent routines are what allow applications to use File Transfer Manager services without regard for the underlying communications protocols. In other words, when an application wants to transfer a file from a remote entity, it tells the File Transfer Manager to get the file and the File Transfer Manager figures out exactly how to implement the transfer for a specific protocol. Multiple instances of the same tool allows the same tool to be used by different processes at the same time, like in a MultiFinder environment, or by different threads in a given application. The file transfer record is described in greater detail later in this chapter.

Besides providing basic file transfer routines, the File Transfer Manager includes routines that make it easy for applications to configure a file transfer tool, either through presenting the user with a dialog box or by interfacing directly with a scripting language. The File Transfer Manager also contains routines that make it easier for you to localize your applications into foreign languages.

You can write applications that use the File Transfer Manager with other parts of Special K to create a communications application with basic connection, terminal emulation, and file transfer capabilities. Or, you can use the Special K File Transfer Manager and substitute some other connection service and terminal emulation service in place of Special K's Connection Manager and Terminal Manager. You can also write your own file transfer tool and add it to the File Transfer Manager. (This procedure is discussed in Chapter 8, "Fundamentals of Writing Your Own Tool.") Regardless of which you choose, your application should be able to handle different file transfer tools such that users can change tools and still be able to use your program.

The file transfer record

The file transfer record describes the file transfer; it contains information like whether to send data or receive data, and where to find the routines that perform the actual sending and receiving of files. The file transfer record also contains pointers to File Transfer Manager internal data structures. Most of the fields in the File Transfer record are filled in when an application calls `FTNew`, which is described later in this chapter.

Because the context for a given file transfer is maintained in a file transfer record, an application can use the same tool more than once at the same time. This allows an application to, amongst other things, perform multiple file transfers (on separate data connections) by creating multiple file transfer records. How to create a file transfer record is described under "Creating a file transfer record" on page nn.

File transfer record data structure

△ **Important** Your application, in order to be compatible with future releases of the File Transfer Manager, should not directly manipulate the fields of the terminal record. The File Transfer Manager provides routines that applications and tools can use to change the fields in the file transfer record. △

```
TYPE
      FTHandle            =     ^FTPtr;
      FTPtr               =     ^FTRecord;
      FTRecord            =     PACKED RECORD

            procID        :     INTEGER;


            flags         :     LONGINT;
            errCode       :     FTErr;

            refCon        :     LONGINT;
            userData      :     LONGINT;

            defProc       :     ProcPtr;
```

```
        config          :       Ptr;
        oldConfig       :       Ptr;

        environsProc    :       ProcPrt;
        reserved1       :       LONGINT;
        reserved2       :       LONGINT;

        Private         :       Ptr;

        SendProc        :       ProcPtr;
        RecvProc        :       ProcPtr;
        WriteProc       :       ProcPtr;
        ReadProc        :       ProcPtr;

        owner           :       WindowPtr;

        Direction       :       INTEGER;
        theReply        :       SFReply;

        WritePtr        :       LONGINT;
        ReadPtr         :       LONGINT;
        TheBuf          :       ^char;
        BufSize         :       LONGINT;
        autoRec         :       Str255;
        attributes      :       INTEGER;
END;
```

procID

`procID` is the file transfer tool ID. This value is dynamically assigned by the Terminal Manager.

flags

`flags` is a bit field that your application can use to determine when a file transfer has completed and if the file transfer was successful. Valid values are:

```
CONST
        ftIsftMode          =       1;
        ftNoMenus           =       2;
        ftQuiet             =       4;
        ftSucc              =       128;
```

`FTIsFTMode` indicates whether or not a file transfer is in progress. A tool will turn this bit on just prior to performing the actual file transfer and will turn it off when the file transfer stops.

`FTNoMenus` indicates that your application should not display any custom menus. This bit is typically used when your application is interfacing with a scripting language.

`FTQuiet` indicates that your application should not display any dialog boxes to alert the user of error conditions. This bit is typically used when your application is interfacing with a scripting language.

`FTSucc` is a bit that is set by the file transfer tool when a file transfer completes successfully.

Your application can first check to see if `FTIsFTMode` toggles from on to off to find out when the file transfer has completed. Then, it can check `FTSucc` to see if the file transfer was completed successfully .

The other bits of `flags` are reserved by Apple Computer, Inc.

errCode

`errCode` contains the last error reported to the File Transfer Manager. If `errCode` is negative, an operating system error occurred. If `errCode` is positive, a File Transfer Manager error occured. Valid values are:

```
CONST
        ftNoErr             =       0;
        ftRejected          =       1;
        ftFailed            =       2;
        ftTimeOut           =       3;
        ftTooManyRetry      =       4;
        ftRemoteCancel      =       6;
        ftWrongFormat       =       7;
        ftNoTools           =       8;
        ftUserCancel        =       9;
```

refCon

`refCon` is a `LONGINT` for use by the application. In a multiple-file-transfer-record environment, `refCon` is used to distinguish one file transfer record from another.

userData

`userData` is a four-byte field that the application can use for any purpose.

defproc

`defproc` is a pointer to the file transfer tool's main definition procedure, which is contained in a code resource of type `fdef`.

config

config is a pointer to a data block that is private to the file transfer tool. It can contain information like retry and timeout values, but this varies from tool to tool. You can find a description of config in Chapter 8, "Fundamentals of Writing Your Own Tool." However, application developers do not need to be concerned with this field; the file transfer tool your application selects will fill in config. To see how this is done, read "Selecting a file transfer tool" on page nn.

oldConfig

oldConfig is a pointer to a data block that is private to the file transfer tool and contains an "old" version of config. This data block is used to implement "undo" operations.

environsProc

environsProc is a pointer to a routine in your application that the file transfer tool can call to obtain a record describing the connection environment. For more information about environsProc, see "Getting connection environment information" on page nn.

reserved1 and reserved2

reserved1 and reserved2 are fields that are reserved for the Terminal Manager.

private

private is a pointer to a data block that is private to the file transfer tool. Your application should not use this field.

SendProc

SendProc is a pointer to a function that the application will use to send data. This function is discussed under "Sending data" on page nn.

RecvProc

RecvProc is a pointer to a function that the application will use to request data . This function is discussed under "Receiving data" on page nn.

WriteProc

WriteProc is a pointer to a function in your application that writes data to a file. The file transfer tool checks this field to see if your application has a WriteProc. If it does, the tool lets the WriteProc handle writing data. If NIL, the file transfer tool performs standard file operations (which is writing to a disk).

This function can be used to perform post-processing upon a file being received; it is discussed under"Writing data" on page nn.

ReadProc

ReadProc is a pointer to a function in your application that reads data from a file. The file transfer tool checks this field to see if your application has a ReadProc. If it does, the tool lets the ReadProc handle writing data. If NIL, the file transfer tool performs standard file operations (which is reading data from a disk).

This function can be used to perform preprocessing upon a file being sent; it is discussed under "Reading data" on page nn.

owner

owner is a pointer to a window, relative to which the file transfer status dialog box is positioned. If this field is NIL, the File Transfer Manager will not display a file transfer status dialog box.

Direction

Direction is a field that indicates whether a file is being sent to or received from another entity. Your application will pass this field as a parameter to FTStart (described on page nn). Valid values in this field are:

```
CONST
      ftReceiving                =      0;
      ftTransmitting             =      1;
```

theReply

theReply is an SFReply data structure. The SFReply data structure should contain both the working directory reference number of the default volume for files being sent or received, as well as the name of the file to be sent (when sending a file) or a file name to use (when receiving a file). If the file transfer protocol already specifies the received file name, pass an empty string for theReply.filename.

WritePtr, ReadPtr, TheBuf, and BufSize

WritePtr, ReadPtr, TheBuf, and BufSize are properties of a particular file transfer tool.

autoRec

autoRec is a string that represents the start sequence that a remote entity sends, causing the Macintosh to enter a file reception mode. If this string is of length 0, remote-entity-initiated file transfers are not supported by the file transfer tool. It is the application's responsibility to make use of this field by searching the data stream for this sequence of characters. The Connection Manager, which is described in Chapter 3, "Connection Manager," provides routines your application can use to search an incomming data stream for a specified sequence of characters.

attributes

`attributes` is a field that describes the file transfer protocol supported by the file transfer tool. The bits in `attributes` are:

```
CONST
      ftSameCircuit             =     1;
      ftSendDisable             =     2;
      ftReceiveDisable          =     4;
      ftTextOnly                =     8;
```

`FTSameCircuit` indicates whether the file transfer tool creates its own data connection or if it expects the application to provide the connection. If this bit is set, the file transfer tool uses the data connection provided by the application. This field is set by the file transfer tool.

`FTSendDisabled` indicates that the file transmission is not supported by the file transfer tool. This bit is used with protocols that do not support sending files and is set by the file transfer tool.

`FTReceiveDisabled` indicates that a file reception is not supported by the file transfer tool. This bit is used with protocols that do not support receiving files and is set by the file transfer tool.

`FTTextOnly` indicates that the file transfer tool can handle sending or receiving only text files (files of type TEXT or data forks); the tool will not handle resource forks . This field is set by the file transfer tool.

The other bits of this field are reserved by Apple Computer, Inc.

File Transfer Manager routines

This section describes the routines that tools and applications can use to access File Transfer Manager services.

```
InitFT / nn
FTNew / nn
FTValidate / nn
FTSetupPreFlight / nn
FTSetupFilter / nn
FTSetupCleanup / nn
FTGetConfig / nn
FTStart / nn
FTAbort / nn
FTDispose / nn
FTResume / nn
FTMenu / nn
FTEnglishToIntl / nn
FTGetRefCon / nn
FTGetUserData / nn
FTGetVersion / nn
ReadProc / nn
RecvProc / nn
MyGetConnEnvirons / nn
FTGetProcID / nn
CMDefault / nn
FTChoose / nn
FTSetupSetup / nn
FTSetupItem / nn
FTSetupPostFlight / nn
FTSetConfig / nn
FTExec / nn
FTCleanup / nn
FTActivate / nn
FTEvent / nn
FTIntlToEnglish / nn
FTSetRefCon / nn
FTSetUserData / nn
FTGetToolName / nn
FTGetFTVersion / nn
SendProc / nn
WriteProc / nn
```

Preparing for a file transfer

Before your application can start a file transfer, itmust first initialize the File Transfer Manager (`InitTM`), find out the `procID` of the tool it requires (`FTGetProcID`), create a file transfer record (`FTNew`), and then configure the file transfer tool (`TMChoose`).

`InitFT` Initializing the File Transfer Manager

`InitFT` initializes the File Transfer Manager. Your application should call this routine after calling the standard Macintosh toolbox initialization routines. If your application uses either the Communications Resource Manager or the Special K Utilities, it should initialize them before initializing the File Transfer Manager.

Procedure

```
InitFT;
```

Description `InitFT` returns an operating system error code if appropriate. If no tools are installed in the File Transfer Manager, it returns `ftNoTools`. Your application is responsible to check for the presence of the Communications Toolbox before calling this function.

`FTGetProcID` Getting current procID information

Your application should call `FTGetProcID` just before creating a new file transfer record to find out the `procID` of a tool.

Function

```
FTGetProcID(name: STR255): INTEGER;
```

Description `name` specifies a file transfer tool. If a file transfer tool exists with the specified `name`, its tool ID is returned. If `name` references a nonexistent file transfer tool, -1 is returned.

`FTNew` Creating a file transfer record

Before your application can transfer files, it must first create a file transfer record. `FTNew` creates a new file transfer record, fills in the fields that it can based upon the parameters that were passed to it, and returns a handle to the new record in `FTHandle`. `FTNew` automatically makes two calls to `FTDefault` (which is described on page nn) to fill in `config` and `oldConfig`. The File Transfer Manager then loads the file transfer tool's main definition procedure, moves it high in the application heap, and locks it. If memory constraints prevent a new file transfer record from being created, `FTNew` passes back `NIL` in `FTHandle`.

Function

```
FTNew(procID: INTEGER; flags: LONGINT; theSendProc:
ProcPtr; theRecvProc: ProcPtr; theReadProc: ProcPtr;
theWriteProc: ProcPtr; theEnvironsProc: ProcPtr;  owner:
WindowPtr; theRefCon: LONGINT; theUserData: LONGINT):
FTHandle;
```

Description

`ProcID` specifies the file transfer tool the File Transfer Manager will use to transfer data.

`flags` is a bit field with the following masks:

```
CONST
    FTIsFTMode          =      1;
    FTNoMenus           =      2;
    FTQuiet             =      4;
    FTSucc              =      128;
```

`FTIsFTMode` indicates whether or not a file transfer is in progress. A tool will turn this bit on just prior to performing the actual file transfer and will turn it off when the file transfer stops.

`FTNoMenus` indicates that your application should not display any custom menus. This bit is typically used when your application is interfacing with a scripting language.

`FTQuiet` indicates that your application should not display any dialog boxes to alert the user of error conditions. This bit is typically used when your application is interfacing with a scripting language.

`FTSucc` is a bit that is set by the file transfer tool when a file transfer completes successfully.

Your application can first check to see if `FTIsFTMode` toggles from on to off to find out when the file transfer has completed. Then, it can check `FTSucc` to see if the file transfer was completed successfully .

The other bits of `flags` are reserved by Apple Computer, Inc.

`theSendProc` is a pointer to a routine that the application uses to send data.

`theRecvProc` is a pointer to a routine that the application uses to request data.

`ReadProc` is a pointer to a routine in your application that reads data from a file. The file transfer tool checks this field to see if your application has a `ReadProc`. If it does, the tool lets the `ReadProc` handle writing data. If `NIL`, the file transfer tool performs standard file operations (which is reading data from a disk).

This function can be used to perform preprocessing upon a file being sent; it is discussed later under "Routines your application provides" on page nn.

`WriteProc` is a pointer to a routine in your application that writes data to a file. The file transfer tool checks this field to see if your application has a `WriteProc`. If it does, the tool lets the `WriteProc` handle writing data. If `NIL`, the file transfer tool performs standard file operations (which is writing to a disk).

This function can be used to perform post-processing upon a file being received; it is discussed later under "Routines your application provides" on page nn.

`environsProc` is a pointer to a routine that your application can call when it wants to get information about the connection. Read Chapter 3, "Connection Manager" to find out about `environsProc`.

`owner` is a pointer to a window, relative to which the file transfer status dialog box is positioned. If this field is `NIL`, the File Transfer Manager will not display a file transfer status dialog box.

`theRefCon` is a `LONGINT` for use by the application. In an environment with more than one file transfer record, an application can use `theRefCon` to distinguish one file transfer record from another.

`theUserData` is the file transfer record reference value field, which the application can use for any purpose.

`CMDefault` Initializing and validating the file transfer record

`CMDefault` fills the specified configuration record with the default configuration specified by the connection tool. This procedure is called automatically by `CMNew` when filling in the `config` and `oldConfig` fields in a new connection record.

Procedure

```
FTDefault(VAR theConfig: Ptr; procID: INTEGER; allocate:
BOOLEAN);
```

Description If `allocate` is `TRUE`, the tool allocates space for `theConfig` in the current zone.

`FTValidate`

`FTValidate` validates the configuration and private data records of the file transfer record by comparing the fields in the file transfer record with the values that are specified in the file transfer tool. This routine is called by `FTNew` and `FTSetConfig` after they have created a new file transfer record to make sure that the the record contains values with those specified by the file transfer tool.

Function

```
FTValidate(hFT: FTHandle): BOOLEAN;
```

Description If the validation fails, `FALSE` is returned and the configuration record is filled with default values.

FTChoose — Configuring a file transfer tool

An application can configure a file transfer tool three ways. The easiest and most straightforward way is by calling the `FTChoose` routine. This routine presents the user with a dialog box similar to the one shown in Figure 5-2. The second way that an application can configure a terminal tool is by presenting the user with a custom dialog box. This method is much more difficult and involves six routines, which are described below (example code is provided in Appendix B, "Useful code samples" to help you implement this feature). The third way that your application can configure a terminal tool is by interfacing directly with a scripting language. This method allows your application to bypass user interface elements.

To present the user with the standard tool-selection dialog box, your application needs to call `FTChoose`.

Function

```
FTChoose(VAR hConn:ConnHandle; where: Point; idleProc:
ProcPtr): INTEGER;
```

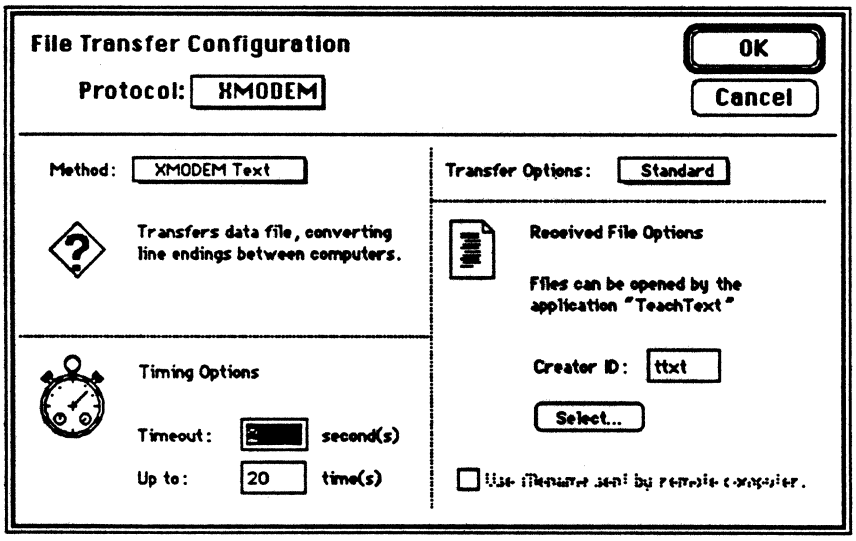
Description

`where` is a point in global coordinates specifying the top-left corner of where the dialog box should appear. It is recommended that your application place the dialog box as close to the top and left of the screen as possible.

`idleProc` is a procedure that the File Transfer Manager will automatically call every time your application loops through the setup dialog box filter procedure.

`FTChoose` will present the user with a dialog box that looks similar to the one shown in Figure 5-2.

- Figure 5-2 The standard tool-selection dialog box



FTChoose will return one of the following values:

```
CONST
      chooseDisaster      =      -2;
      chooseFailed        =      -1;
      chooseAborted       =      0;
      chooseOKMinor       =      1;
      chooseOKMajor       =      2;
      chooseCancel        =      3;
```

chooseDisaster means that the choose operation failed and destroyed the file transfer record.

chooseFailed means that the choose operation failed and the file transfer record was not changed.

chooseAborted means that the user tried to change the file transfer method while the transfer was still in progress, thereby failing to complete the FTChoose operation.

chooseOKMinor means that the user selected OK in the dialog box, but did not change the file transfer tool being used.

chooseOKMajor means that the user selected OK in the dialog box and also changed the file transfer tool being used. The old file transfer handle is destroyed by the File transfer Manager, by calling FTDispose (the file transfer is closed down and all pending reads and writes are terminated) and a new file transfer handle is returned in hConn.

chooseCancel means that the user selected Cancel in the dialog box.

Custom configuration of a file transfer

To present the user with a custom tool-configuration dialog box, your application needs to call a series of six File Transfer Manager routines: FTSetupPreflight, FTSetupSetup, FTSetupItem, FTSetupFilter, FTSetupCleanup, and FTSetupPostflight. Using these routines is more involved than calling FTChoose, but they provide your application with much more flexibility. There is a code sample in Appendix B, "Useful code samples" that shows how an application calls these routines.

To build a list of file transfer tools, use the routine CRMGetIndToolName, which is described in Chapter 4, "Communications Resource Manager."

FTSetupPreflight

FTSetupPreflight returns a handle to a dialog item list from the file transfer tool that your application should append to the configuration dialog box. (Your application can use AppendDITL which is discussed in Chapter 7, "Special K Utilities.")

The file transfer tool can use FTSetupPreflight to allocate a block of private storage, and to store the pointer to that block in magicCookie. This value, magicCookie, should be passed to the other routines that are used to setup the configuration dialog box.

Function

```
FTSetupPreflight(procID: INTEGER; VAR magicCookie:
LONGINT): Handle;
```

Description

procID is the ID for the file transfer tool that is being configured. Your application should get this value by using the FTGetProcID routine, which is discussed later in this chapter.

The refcon of the custom dialog box should point to the procID of the tool being configured.

magicCookie should be passed to the other routines that your application uses to setup the configuration dialog box

FTSetupSetup

FTSetupSetup tells the file transfer tool to set up controls (like radio buttons or check boxes) in the dialog item list returned by FTSetupPreflight .

Procedure

```
FTSetupSetup(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR magicCookie: LONGINT);
```

Description

procID is the ID for the file transfer tool being configured. Your application should get this value by using the FTGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

magicCookie is the value returned from the FTSetupPreflight.

FTSetupFilter

FTSetupFilter should be called as a filter procedure prior to the standard modal dialog box filter procedure for the configuration dialog box. This routine allows file transfer tools to filter events in the configuration dialog box.

Function

```
FTSetupFilter(procID: INTEGER; theConfig: Ptr;
count:INTEGER; theDialog: DialogPtr; VAR theEvent:
EventRecord; VAR theItem: INTEGER; VAR magicCookie:
LONGINT): BOOLEAN;
```

Description

procID is the ID for the file transfer tool that is being configured. Your application should get this value by using the FTGetProcID routine, which is discussed later in this chapter.

count is the number of the first item in the dialog item list appended to the dialog box.

theConfig is the pointer to the configuration record for the tool being configured.

theDialog is the dialog box performing the configuration.

theEvent is the event record for which filtering is to take place.

theItem can return the appropriate item clicked on in the dialog box.

magicCookie is the value returned from FTSetupPreflight.

If the event passed in was handled, TRUE is returned. Otherwise, FALSE indicates that standard dialog box filtering should take place.

FTSetupItem

FTSetupItem processes mouse events for controls in the custom configuration dialog box.

Procedure

```
FTSetupItem(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR item: INTEGER; VAR
magicCookie: LONGINT);
```

Description

procID is the ID for the file transfer tool being configured. Your application should get this value by using the FTGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

item is the item clicked on in the dialog box. This value can be modified and sent back.

magicCookie is the value returned from FTSetupPreflight.

FTSetupCleanup

FTSetupCleanup disposes of any storage allocated in FTSetupPreflight and performs any other clean-up operation.

Procedure

```
FTSetupCleanup(procID: INTEGER; theConfig: Ptr; count:
INTEGER; theDialog: DialogPtr; VAR magicCookie: LONGINT);
```

Description

procID is the ID for the file transfer tool that is being configured. Your application should get this value by using the FTGetProcID routine, which is discussed later in this chapter.

theConfig is the pointer to the configuration record for the tool being configured.

count is the number of the first item in the dialog item list appended to the dialog box.

theDialog is the dialog box performing the configuration.

magicCookie is the value returned from FTSetupPreflight.

FTSetupPostflight

FTSetupPostflight either shortens or disposes of the dialog box. It will close the tool file if it is not being used by any other sessions.

Procedure

```
FTSetupPostflight(procID:INTEGER);
```

Description

procID is the ID for the file transfer tool that is being configured. Your application should get this value by using the FTGetProcID routine, which is discussed later in this chapter.

Scripting language interface

Your application does not have to rely on a user making selections from dialog boxes in order to configure a file transfer tool. FTGetConfig and FTSetConfig provide the function that your application needs to interface with a scripting language.

FTGetConfig

FTGetConfig returns a null-terminated string (an example of which is shown after the description of the next routine) containing tokens that fully describe the configuration of the file transfer record.

Function

```
FTGetConfig(hFT: FTHandle): Ptr;
```

Description

If an an error occurs, FTGetConfig will return NIL. It is the responsibility of your application to dispose of Ptr.

FTSetConfig

FTSetConfig passes a null-terminated string to the file transfer tool for parsing (an example of which is shown under "A sample null-terminated configuration string") that is pointed to by thePtr. The string, which can be any length, must contain tokens that describe the configuration of the file transfer record, and is parsed from left to right.

Function

```
FTSetConfig(hFT: FTHandle; thePtr: Ptr): INTEGER;
```

Description

Items that are not recognized or relevant are ignored; this causes FTSetConfig to abort parsing the string and to return the character position where the error occurred. If parsing is successfully completed, FTSetConfig returns noErr. FTSetConfig may also return -1 to indicate a general problem with processing the configuration string..

The parsing operation is the responsibility of the individual tool.

A sample null-terminated configuration string

```
InterCharDelay 0 InterLineDelay 0 WordWrap
False Ending CR \0
```

Transferring files

Once your application has performed all of the necessary preparation for a file transfer calling initFT, creating a file transfer record with FTNew, and choosing a file transfer tool with FTChoose (or equivalent means), it is ready to start transfering files. There are two steps that your application must take to do this: first, it must call FTStart to open the file and initialize tool-private variables; second, it must call FTExec to process a packet of data every time it goes through its main event loop.

FTStart

Besides opening the file that is going to be involved in the file transfer and initializing tool-private variables, `FTStart` also sends or receives the file transfer's first packet.

The appearance of a status dialog box is controlled by the value in `owner` in the file transfer record.

The code that controls the actual sending, receiving, reading, and writing of data is the responsibility of your application. Your application specifies these procedures when it creates the file transfer record. A description of the parameters that will be passed to these routines is discribed later in this chapter under "Your application's send, receive, read, and write functions" on page nn.

Function

```
FTStart(hFT: FTHandle, direction:INTEGER,
fileInfo:SFReply): FTErr;
```

Description

Once the file transfer has started, your application needs to call `FTExec` every time that it goes through its main event loop. This gives the tool time to send and receive a packet of data.

There is sample code in Appendix B, "Useful code samples" that shows you an effective strategy for calling `FTExec`.

FTExec

Your application should call FTExec every time it goe through its main event loop to give the file transfer tool time to send and receive data.

Procedure

```
FTExec(hFT: FTHandle);
```

FTAbort

Stopping a file transfer

`FTAbort` aborts a file transfer in progress.

Function `FTAbort(hFT: FTHandle): FTErr;`

`FTDispose` Disposing of a file transfer record

`FTDispose` disposes of the file transfer record and all associated data structures. Any file transfer in progress (as specified by the file transfer record) is aborted.

Procedure `FTDispose(hFT: FTHandle);`

`FTCleanup` Cleaning up after a file transfer

`FTCleanup` is not typically called by an application. The File Transfer Manager uses this call internally to close the file and to deallocate the memory that had been allocated for the file transfer.

Function `FTCleanup(hFT: FTHandle; now: BOOLEAN): FTErr;`

Handling events

The File Transfer Manager event processing routines provide useful extensions to the Macintosh Toolbox Event Manager. The section below explains the three procedures that Special K provides: `FTActivate`, `FTResume`, and `FTEvent`. There is example code in Appendix B, "Useful code samples" that shows how an application can determine if an event needs to be handled by one of these routines.

`FTActivate` Activate events

`FTActivate` processes an activate or deactivate event (for instance, installing or removing a custom tool menu) for a window that the file transfer is associated with.

Procedure `FTActivate(hFT: FTHandle; act: BOOLEAN);`

Description If `act` is `TRUE`, an activate event is to be processed. Otherwise, a deactivate event is to be processed.

FTResume — Resume events

FTResume is called in response to the application receiving a suspend or a resume event. The file transfer tool may decide to change timeout values or other parameters depending on whether or not the application is in the foreground.

Procedure

```
FTResume(hFT: FTHandle; res: BOOLEAN);
```

FTMenu — Menu events

Your application should call FTMenu in response to a menu selection from a menu that is installed by the file transfer tool.

Function

```
FTMenu(hFT: FTHandle; menuID: INTEGER; item: INTEGER):
BOOLEAN;
```

Description

FTMenu returns FALSE if the menu item was not handled by the file transfer tool. FTMenu returns TRUE if the file transfer tool did handle the menu item.

FTEvent — Other events

FTEvent is called in response to receiving an event for a window that belongs to the file transfer tool. Windows (or dialog boxes) that belong to the File Transfer Manager should have a file transfer handle stored in refCon of windowRecord.

Procedure

```
FTEvent(hFT: FTHandle; theEvent: EventRecord);
```

Localizing strings

Special K provides two routines that make it easier to localize strings.

FTIntlToEnglish

FTIntlToEnglish converts a configuration string that is pointed to by inputPtr in the given language to an American English configuration string that is pointed to by outputPtr.

Function

```
FTIntlToEnglish(hFT: FTHandle; inputPtr: Ptr; VAR
outputPtr: Ptr; language: INTEGER): OSErr;
```

Description

`language` specifies the language from which the string is to be converted.

The file transfer tool allocates space for `outputPtr`.

If the language specified is not supported, `noErr` is still returned, but `outputPtr` is `NIL`.

The function returns an operating system error code if any internal errors occur.

FTEnglishToIntl

`FTEnglishToIntl` converts a configuration string that is pointed to by `inputPtr` in English to a configuration string in the given language that is pointed to by `outputPtr`.

Function

```
FTEnglishToIntl(hFT: FTHandle; inputPtr: Ptr; VAR
outputPtr: Ptr; language: INTEGER): OSErr;
```

Description

`language` specifies the language to which the string is to be converted.

The file transfer tool allocates space for `outputPtr`.

If the language specified is not supported, `noErr` is still returned, but `outputPtr` is `NIL`.

The function returns an operating system error code if any internal errors occur.

Miscellaneous routines

FTSetRefCon

`FTSetRefCon` sets the file transfer record's reference constant to the given value.

Procedure

```
FTSetRefCon(hFT: FTHandle; rC: LONGINT);
```

FTGetRefCon

`FTGetRefCon` returns the file transfer record's reference constant.

Function `FTGetRefCon(hFT: FTHandle): LONGINT;`

FTSetUserData

`FTSetUserData` sets the file transfer record's `userData` field to the given value. It is very important that your application uses this routine to change the value of the `userData` field instead of changing it directly.

Procedure `FTSetUserData(hFT: FTHandle; uD: LONGINT);`

FTGetUserData

`FTGetUserData` returns the file transfer record's `userData` field.

Function `FTGetUserData(hFT: FTHandle) : LONGINT;`

FTGetToolName

`FTGetToolName` returns in `name` the name of the tool specified by `procID` If `procID` references a file transfer tool that does not exist, the File Transfer Manager will set `name` to an empty string.

Procedure `FTGetToolName(procID: INTEGER; VAR name: STR255);`

FTGetVersion

`FTGetVersion` returns a handle to a relocatable block that contains the same information as is contained in the `fver` resource. The file transfer tool must have the same ID as that specified by `FTHandle`. This handle is *not* a resource handle.

Function `FTGetVersion(hFT: FTHandle): Handle;`

FTGetFTVersion

`FTGetFTVersion` returns the version number of the File Transfer Manager.

Function `FTGetFTVersion: INTEGER;`

Routines your application provides

Your application is responsible for providing routines it will use to read, send, receive, and write data during a file transfer. Your application might also need to provide a routine that can provide information to the file transfer tool about the connection environment. When your application creates a new file transfer record, it specifies pointers to these routines.

Both sending and receiving files is a two-step process. In the case of sending a file, your application must first read the data from a file into a buffer (with `ReadProc`) and then send it to the remote application (with `SendProc`). In the case of receiving a file, your application must first read the remote application's data into a buffer (with `RecvProc`) and then write it to disk (with `WriteProc`).

Of the five routines below, your application must include the send and receive routines. The other routines are optional.

`ReadProc` Reading data

Function

```
ReadProc(VAR count: LONGINT; bufPtr: Ptr; refCon:
LONGINT): OSErr;
```

Description

`count` is the number of bytes to read into the buffer. After `ReadProc` has completed, `count` must contain the actual number of bytes that were read.

`bufPtr` points to the buffer into which the data was read.

`refCon` is the reference constant of the file transfer record.

`ReadProc` must return an error code when appropriate.

`SendProc` Sending data

Function

```
SendProc(thePtr: Ptr; theSize: LONGINT; refCon: LONGINT;
channel: INTEGER):LONGINT;
```

Description

`thePtr` is a pointer to a block of data in memory that is to be sent.

`theSize` is the length of that block.

`refCon` is the reference constant of the file transfer record.

`channel` specifies the data channel that the file transfer tool can use. Your application should specify one of the following values for channel: `CMData`, `CMCntl`, or `CMAttn`.

`SendProc` must return the actual number of bytes that were sent.

Sample send routine

```
FUNCTION FTSendProc(thePtr : Ptr; theSize : LONGINT;
                                                    refcon : LONGINT; channel:
INTEGER; flags: INTEGER) : LONGINT;
VAR
      theWindow : WindowPtr;
      pWindow : WindowP;

      theErr : CMErr;
      EOM : BOOLEAN;

BEGIN
      theWindow := WindowPtr(refcon);
      theConn:= GethConn(theWindow);
      FTSendProc := 0;
      IF theConn= NIL THEN
            Exit(FTSendProc);

      IF WindowPeek(theWindow)^.windowKind <> userKind THEN
            Exit(FTSendProc);

      pWindow := WindowP(GetWRefCon(theWindow));
      theTerm := pWindow^.hTerm;
      theConn := pWindow^.hConn;

      theErr := CMWrite(theConn, thePtr, theSize, channel, FALSE, NIL, 0,
flags);
      FTSendProc := theSize;
END;
```

`RecvProc` Receiving data

Function

```
RecvProc(thePtr: Ptr; theSize: LONGINT; refCon: LONGINT;
channel: INTEGER):LONGINT;
```

Description

`thePtr` is a pointer to a block of data in memory where the incomming data is to be placed.

`theSize` is the length of that data.

`refCon` is the reference constant of the file transfer record.

`channel` specifies the data channel that the file transfer tool can use. Your application should specify one of the following values for channel: `CMData`, `CMCntl`, or `CMAttn`.

`RecvProc` must return the actual number of bytes received.

Sample receive routine

```
FUNCTION FTReceiveProc(thePtr : Ptr; theSize : LONGINT; refcon : LONGINT;
                                                    channel: INTEGER; VAR
flags: INTEGER) : LONGINT;
VAR
      theWindow : WindowPtr;
      pWindow : WindowP;

      theErr : CMErr;
      EOM : BOOLEAN;

BEGIN
      theWindow := WindowPtr(refcon);
      theConn:= GethConn(theWindow);
      FTReceiveProc := 0;
      IF theConn= NIL THEN
            Exit(FTReceiveProc);

      IF WindowPeek(theWindow)^.windowKind <> userKind THEN
            Exit(FTReceiveProc);

      pWindow := WindowP(GetWRefCon(theWindow));
      theTerm := pWindow^.hTerm;
      theConn := pWindow^.hConn;

      EOM := TRUE;
      theErr := CMRead(theConn, thePtr, theSize, channel, FALSE, NIL, 0,
flags);
      FTReceiveProc := theSize;
END;
```

`WriteProc` Writing data

Function

```
WriteProc(VAR count: LONGINT; bufPtr: Ptr; refCon:
LONGINT): OSErr;
```

Description

`count` is the number of bytes to write. After `WriteProc` has completed, `count` must contain the actual number of bytes that was written.

`bufPtr` is a pointer to the data in memory.

`refCon` is the reference constant of the file transfer record.

`WriteProc` must return an error code when appropriate.

MyGetConnEnvirons Getting connection environment information

Your application might need to pass information about the connection environment to the file transfer tool. To accomplish this, the file transfer tool calls a routine in the application, `MyGetConnEnvirons`.

Function

```
MyGetConnEnvirons(refCon: LONGINT; VAR theEnvirons:
ConnEnvironRec): CMErr;
```

Description

`refCon` is the reference constant for the file transfer tool

`theEnvirons` is a data structure containing the connection environment record. Your application can either construct `theEnvirons` or use the Connection Manager routine `CMGetConnEnvirons`. For more information about `theEnvirons`, read Chapter 3, "Connection Manager."

Sample connection environment routine

```
FUNCTION FTGetConnEnvirons(refCon: LONGINT; VAR theEnvirons:
ConnEnvironRec): OSErr;
VAR
      theWindow : WindowPtr;
      pWindow : WindowP;

BEGIN
      theWindow := WindowPtr(refcon);
      theConn:= GethConn(theWindow);

      FTGetConnEnvirons := envNotPresent;
                  { pessimism }
      IF theConn= NIL THEN
            Exit(FTGetConnEnvirons);

      IF WindowPeek(theWindow)^.windowKind <> userKind THEN
            Exit(FTGetConnEnvirons);

      pWindow := WindowP(GetWRefCon(theWindow));
      theConn := pWindow^.hConn;

      FTGetConnEnvirons := CMGetConnEnvirons(theConn, theEnvirons);
END;
```

Summary

File Transfer Manager routines	*see page*
`FTAbort(hFT: FTHandle): FTErr;`	nn
`FTActivate(hFT: FTHandle; act: BOOLEAN);`	nn
`FTChoose(VAR hFT: FTHandle; where: Point; idleProc: ProcPtr): INTEGER;`	nn
`FTCleanup(hFT: FTHandle; now: BOOLEAN): FTErr;`	nn
`FTDefault(VAR theConfig: Ptr; procID: INTEGER; allocate: BOOLEAN);`	nn
`FTDispose(hFT: FTHandle);`	nn
`FTEnglishToIntl(hFT: FTHandle; inputPtr: Ptr; VAR outputPtr: Ptr; language: INTEGER): INTEGER;`	nn
`FTEvent(hFT: FTHandle; theEvent: EventRecord);`	nn
`FTExec(hFT: FTHandle);`	nn
`FTGetConfig(hFT: FTHandle): Ptr;`	nn
`FTGetFTVersion: INTEGER;`	nn
`FTGetName(procID: INTEGER; VAR name: STR255);`	nn
`FTGetProcID(name: STR255): INTEGER;`	nn
`FTGetRefCon(hFT: FTHandle): LONGINT;`	nn
`FTGetUserData(hFT: FTHandle) : LONGINT;`	nn
`FTGetVersion(hFT: FTHandle): Handle;`	nn
`FTIntlToEnglish(hFT: FTHandle; inputPtr: Ptr; VAR outputPtr: Ptr; language: INTEGER): INTEGER;`	nn
`FTMenu(hFT: FTHandle; menuID: INTEGER; item: INTEGER): BOOLEAN;`	nn

```
FTNew(procID: INTEGER; flags: LONGINT; theSendProc:          nn
              ProcPtr; theRecvProc: ProcPtr; theReadProc:
              ProcPtr; theWriteProc: ProcPtr;
              theEnvironsProc: ProcPtr;  owner: WindowPtr;
              theRefCon: LONGINT; theUserData: LONGINT):
              FTHandle;

FTResume(hFT: FTHandle; res: BOOLEAN);                       nn

FTSetConfig(hFT: FTHandle; thePtr: Ptr): INTEGER;            nn

FTSetRefCon(hFT: FTHandle; rC: LONGINT);                     nn

FTSetupCleanup(procID: INTEGER; theConfig: Ptr; count:       nn
              INTEGER; theDialog: DialogPtr; VAR
              magicCookie: LONGINT);

FTSetupFilter(procID: INTEGER; theConfig: Ptr; count:        nn
              INTEGER; theDialog: DialogPtr; VAR theEvent:
              EventRecord; VAR theItem: INTEGER; VAR
              magicCookie: LONGINT): BOOLEAN;

FTSetupItem(procID: INTEGER; theConfig: Ptr; count:          nn
              INTEGER; theDialog: DialogPtr; VAR theItem:
              INTEGER; VAR magicCookie: LONGINT);

 FTSetupPostflight(procID:INTEGER);                          nn

FTSetupPreflight(procID: INTEGER; VAR magicCookie:           nn
              LONGINT): Handle;

FTSetupSetup(procID: INTEGER; theConfig: Ptr; count:         nn
              INTEGER; theDialog: DialogPtr; VAR
              magicCookie: LONGINT);

FTSetUserData(hFT: FTHandle; uD: LONGINT);                   nn

FTStart(hFT: FTHandle): FTErr;                               nn

FTValidate(hFT: FTHandle): BOOLEAN;                          nn

InitFT;                                                      nn
```

Routines in your application *see page*

```
MyGetConnEnvirons(refCon: LONGINT; VAR theEnvirons:          nn
            ConnEnvironRec): CMErr;

ReadProc(VAR count: LONGINT; bufPtr: Ptr; refCon:            nn
            LONGINT):OSErr;

RecvProc(thePtr: Ptr; theSize: LONGINT; refCon: LONGINT;     nn
            channel: INTEGER):LONGINT;

SendProc(thePtr: Ptr; theSize: LONGINT; refCon: LONGINT;     nn
            channel: INTEGER):LONGINT;

WriteProc(VAR count: LONGINT; bufPtr: Ptr; refCon:           nn
            LONGINT): OSErr;
```

Data types

File transfer record

```
TYPE
      FTHandle          =     ^FTPtr;
      FTPtr             =     ^FTRecord;
      FTRecord          =     PACKED RECORD

            procID      :     INTEGER;


            flags       :     LONGINT;
            errCode     :     FTErr;

            refCon      :     LONGINT;
            userData    :     LONGINT;

            defProc     :     ProcPtr;

            config      :     Ptr;
            oldConfig   :     Ptr;

            environsProc :    ProcPrt;
            reserved1   :     LONGINT;
            reserved2   :     LONGINT;

            Private     :     Ptr;
```

```
        SendProc        :       ProcPtr;
        RecvProc        :       ProcPtr;
        WriteProc       :       ProcPtr;
        ReadProc        :       ProcPtr;

        owner           :       WindowPtr;

        Direction       :       INTEGER;
        theReply        :       SFReply;

        WritePtr        :       LONGINT;
        ReadPtr         :       LONGINT;
        TheBuf          :       ^char;
        BufSize         :       LONGINT;
        autoRec         :       Str255;
        attributes      :       INTEGER;
    END;
```

Constants

```
CONST
{ send & receive }
        ftReceiving                 =       0;
        ftTransmitting              =       1;

{ file transfer attributes }
        ftSameCircuit               =       1;
        ftSendDisable               =       2;
        ftReceiveDisable            =       4;
        ftTextOnly                  =       8;

{ file transfer flags }
        ftIsftMode          =       1;
        ftNoMenus           =       2;
        ftQuiet             =       4;
        ftSucc              =       128;

{ Choose return values }
        chooseDisaster      =       -2;
        chooseFailed        =       -1;
        chooseAborted       =       0;
        chooseOKMinor       =       1;
        chooseOKMajor       =       2;
        chooseCancel        =       3;
```

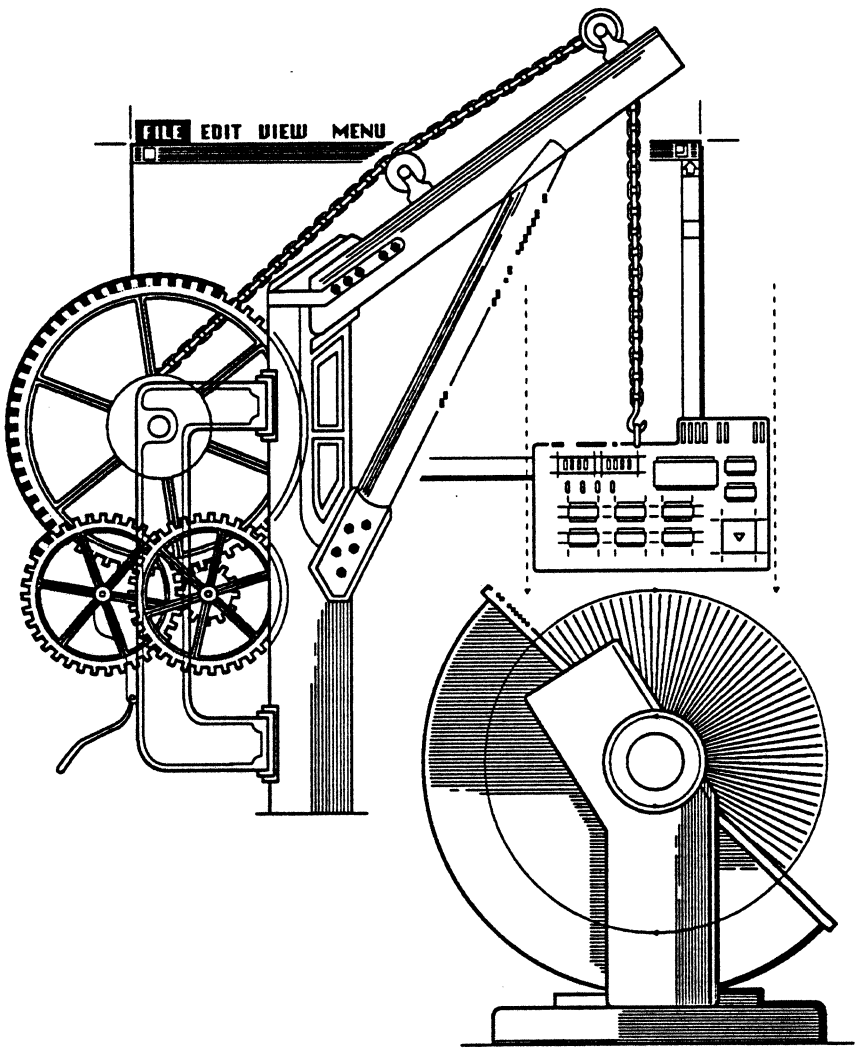
Error codes

```
CONST
        ftNoErr             =       0;
        ftRejected          =       1;
        ftFailed            =       2;
        ftTimeOut           =       3;
        ftTooManyRetry      =       4;
        ftRemoteCancel      =       6;
        ftWrongFormat       =       7;
        ftNoTools           =       8;
        ftUserCancel        =       9;
```

File Transfer Manager Routine Selectors

```
InitFT              .EQU    513         FTSetRefCon         .EQU    514

FTGetRefCon         .EQU    515         FTSetUserData       .EQU    516

FTGetUserData       .EQU    517         FTGetToolName       .EQU    518

FTGetProcID         .EQU    519         FTNew               .EQU    520

FTDispose           .EQU    521         FTExec              .EQU    522

FTStart             .EQU    523         FTCleanup           .EQU    524

FTAbort             .EQU    525         FTResume            .EQU    526

FTValidate          .EQU    527         FTDefault           .EQU    528

FTSetupPreflight    .EQU    529         FTSetupSetup        .EQU    530

FTSetupFilter       .EQU    531         FTSetupItem         .EQU    532

FTSetupCleanup      .EQU    533         FTGetConfig         .EQU    534

FTSetConfig         .EQU    535         FTIntlToEnglish     .EQU    536

FTEnglishToIntl     .EQU    537         FTGetVersion        .EQU    538

FTGetFTVersion      .EQU    539         FTChoose            .EQU    540

FTEvent             .EQU    541         FTSetupPostflight   .EQU    542

FTMenu              .EQU    543         FTActivate          .EQU    544
```

Chapter 6 **Communications Resource Manager**



About this chapter

This chapter describes the Communications Resource Manager, which is the Special K manager that makes it easier for your application to manage devices (like internal modems and serial cards) and resources.

After an introduction to the Communications Resource Manager, this chapter describes the data structures and routines your application can use to implement device management. Then it describes the routines your application can use to perform resource management. A summary at the end provides a quick reference to these features.

Often refered to in this chapter is the term "your application", which is the application you are writing for the Macintosh, and which will implement communication services for users. Be careful not to confuse the services your application is requesting with the services that tools provide.

In order to use the Communications Resource Manager, you need to be familiar with the following topics:

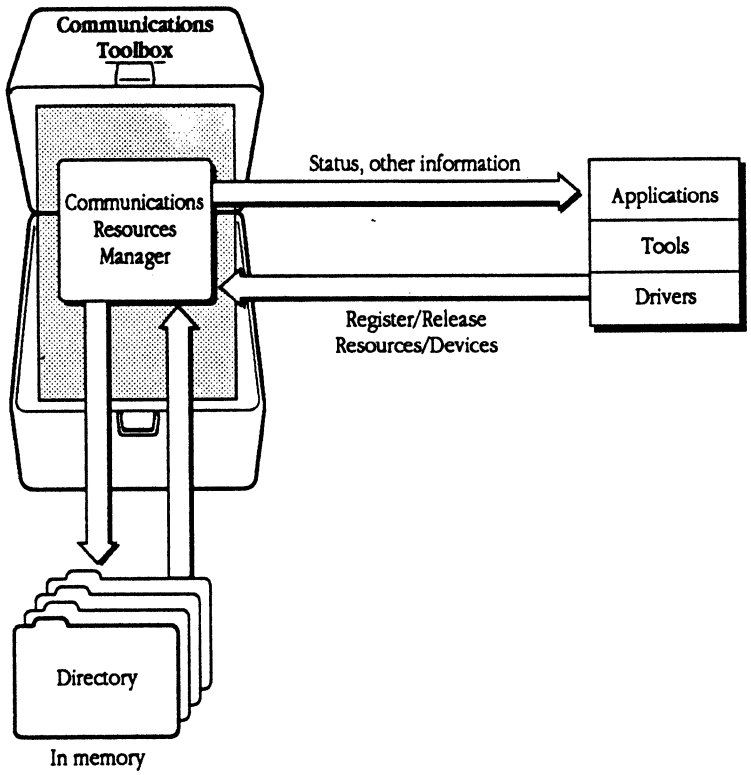
- Resource Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Device Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Memory Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Operating System Utilities (see *Inside Macintosh*, Volume II)
- MultiFinder programming environment <<<anyone have a book to reference for this one? >>>

About the Communications Resource Manager

There are two reasons an application would use the services provided by the Communications Resource Manager: to manage devices and to manage resources. Device management is essential when your application needs to know about new cards that have been installed in a Macintosh. Resource management is required when your application is sharing resources with other applications (as it does when a Macintosh runs MultiFinder). The resource management services provided in the Communications Resource Manager are an extension to the services provided by the Resource Manager in the Macintosh Toolbox. Although the tasks of device management and resource management are somewhat different in nature, the routines and data structures your application needs to perform these tasks are provided in one place, the Communications Resource Manager.

The way your application interfaces with the Communications Resource Manager is very similar to the way it interfaces with other Special K managers. Your application calls a Communications Resource Manager routine, which upon completion, returns to your application any relevant parameters and return codes. Figure 6-1 shows the data flow into and out of the Communications Resource Manager.

Figure 6-1: Data flow in and out of the Communications Resource Manager.



Device management

The way Macintosh applications interact with special interface cards varies from card to card, making the task of programming the Macintosh to use these cards quite complex. Special K solves this problem by providing applications with standardized routines and data structures that your application can use to keep track of communications devices (typically in the form of add-in cards) that users have installed.

The data structure that is most important in supporting communications device management is the Communications Resource Record, which is stored as a linked list. The Communications Resource Record is comprised of fields containing information like what type of device the record represents and whether or not the device is available for use and is described on page nn.

The Communications Resource Manager and your application keep track of communications devices by placing a Communications Resource Record into the linked list for each communications device. Initially, when your application calls `InitCRM` (which is discussed later under "Initializing the Communications Resource Manager" on page nn), this linked-list contains two elements, one for each of the serial drivers. Your application then adds and deletes communications resource records.

By making use of Special K routines, your application can register new devices, allocate devices, and look for specific kinds of devices. Also device drivers, if properly coded, can resolve conflicts that arise out of the need for more than one application to use the same communications resource at the same time. The need to contend with this type of problem is becomming increasingly common as more users run under MultiFinder.

Resource management

When you application shares resources with other applications, problems can arise if one of the sharing applications accidentally ruins or disposes of a resource that is needed by another application. Special K provides routines that your application can use to share resources and avoid this kind of problem. These routines index (or number) resources as they are requested. Then, when your application releases a resource, it can specify the index on it's copy of the resource. This enables the Communications Resource Manager to keep track of which resources are still being used.

The communications resource record

The most important data structure to the Communications Resource Manager is the communications resource record. It is so important because it contains information like the name and type of devices and whether or not a device is in use.

At start-up time, the Communications Resource Manager builds a linked-list of communications resource records. If the Communications Resource Manager is installed, the linked-list will consist of a minimum of two devices of type `crmSerialDevice`.

When your application installs a new record into the linked-list, it must fill in all the fields of the communications resource record with one exception, `CRMDeviceID`. The Communications Resource Manager fills in this field.

The communications resource record data sructure

```
TYPE
      CRMRecPtr            = ^CRMRec;
      CRMRec               = RECORD
            qLink                :     QElemPtr;
            qType                :     INTEGER;
            crmVersion           :     INTEGER;

            crmPrivate           :     LONGINT;
            crmReserved          :     INTEGER;

            crmDeviceType        :     LONGINT;
            crmDeviceID          :     LONGINT;
            crmAttributes        :     LONGINT;
            crmStatus            :     LONGINT;

            crmRefCon            :     LONGINT;
      END;
```

qLink

qLink points to the next CRMRec in the Communications Resource Manager's linked-list of communications resource records..

qType

qType is a constant that your application must fill with a 9.

crmVersion

crmVersion is the version number of the CRMRec data structure. At this time there is only one version, so your application must fill this field in with a 1.

crmPrivate and crmReserved

crmPrivate and crmReserved are private to the Communications Resource Manager; your application must not use them for any reason.

crmDeviceType

crmDeviceType is the type of device (for example, a serial port).

crmDeviceID

crmDeviceID is an identifier that your application can use to distinguish between multiple devices of the same device type. The Communications Resource Manager fills in this field when your application calls the CRMInstall routine

crmAttributes

crmAttributes specifies the attributes of a specific device type. This field can hold either a pointer to the data or the actual data that describes the device. There is a sample crmAttributes data sturcture for a serial device on page nn.

crmStatus

crmStatus specifies the status of a device. Your application can use this field for device arbitration purposes.

crmRefCon

crmRefCon is available for rent.

Communications Resource Manager routines

This section describes the routines that applications use to access Communications Resource Manager services.

```
InitCRM / nn
CRMSearch / nn
CRMGetCRMVersion / nn
CRMGetResource / nn
CRMGetIndResource / nn
CRMGetNamedResource / nn
CRMGetIndex / nn
CRMGetIndToolName / nn
CRMRealToLocalID / nn
CRMInstall / nn
CRMRemove / nn
CRMGetHeader / nn
CRMGet1Resource / nn
CRMGet1IndResource / nn
CRMGet1NamedResource / nn
CRMReleaseResource / nn
CRMLocalToRealID / nn
```

InitCRM — Initializing the Communications Resource Manager

`InitCRM` initializes the Communications Resource Manager. This function should be called after calling the standard toolbox initialization routines and before any of the other Special K manager initialization routines.

Function

```
InitCRM:OSErr;
```

Description `InitCRM` will return an operating system error code if appropriate. Your application should check for the presence of the Communications Toolbox before calling this function.

CRMInstall — Installing devices

`CRMInstall` installs a device into the Communications Resource Manager's linked-list. Devices in the Communications Resource Manager's linked-list typically have their `CRMRecs` allocated in the system heap. If your application installs a `CRMRec` at start-up time, be sure that it increases the size of the system heap appropriately. If your application installs a `CRMRec` during run-time, make sure that enough system heap space is available, or use the application heap.

For more information on how to register a device with the Communications Resource Manager, read "How to Register a Device" on page nn.

Procedure

```
CRMInstall(TheCRMReqPtr: QElemPtr);
```

Description `CRMInstall` returns in `TheCRMReqPtr` a pointer to the new communications resource record.

▲ **Warning** `CRMRecs` allocated in the application heap need to be removed prior to reinitialization of the application heap; otherwise, the Communications Resource Manager's linked-list may be damaged. ▲

CRMSearch — Searching for devices

Your application can use `CRMSearch` to order the Communications Resource Manager's linked-list, or to add new elements on to the end of the linked-list.

Function `CRMSearch(crmReqPtr: QElemPtr): QElemPtr;`

Description This function takes the `CRMRec` pointed to by `crmReqPtr` and searches for a device in the Communications Resource Manager's linked-list that has two characteristics: the same `deviceType` and a `deviceID` greater than the `deviceID` in the specified record. `CRMSearch` will return a pointer to the first record that it finds that meets these two conditions. Or, if no records meet the search criteria, it will return `NIL`.

When searching for the first element in the linked-list, your application must pass 0 in `deviceID`.

`CRMRemove` Removing devices

`CRMRemove` removes a device from the Communications Resource Manager's linked-list.

Function `CRMRemove(crmReqPtr: QElemPtr): OSErr;`

Description `crmReqPtr` specifies the device to be removed..

`CRMGetVersion`

`CRMGetVersion` returns the version of the Communications Resource Manager.

Function `CRMGetCRMVersion:INTEGER;`

`CRMGetHeader`

`CRMGetHeader` returns a pointer to the head of the Communications Resource Manager's linked-list.

Function `CRMGetHeader: QHdrPtr;`

Resource management routines

The eight routines that are described below make it easier for your application to manage communications resources. Your application should use these routines so that the Communications Resource Manager can maintain a list that, among other things, indicates how many times a resource is simultaneously in use.

You will probably recognize the names of these routines because they are so similar to the names of Resource Manager routines available in the Macintosh Toolbox. They also operate very similarly; in fact, most of these routines make use of their similarly-named counterparts in the Macintosh Toolbox.

`CRMGetResource` and `CRMGet1Resource`

`CRMGetResource` and `CRMGet1Resource` call `GetResource` and `Get1Resource` respectively, and return a handle to the specified communications resource. The Communications Resource Manager then adds the handle to the list of resources that it is managing and increases by one the value that indicates how many times a resource is simultaneously in use.

Function `CRMGetResource(theType: ResType; theID: INTEGER): Handle;`

Function `CRMGet1Resource(theType: ResType; theID: INTEGER): Handle;`

`CRMGetIndResource` and `CRMGet1IndResource`

`CRMGetIndResource` and `CRMGet1IndResource` call `GetIndResource` and `Get1IndResource` respectively, and return a handle to the specified communications resource. The Communications Resource Manager then adds the handle to the list of resources that it is managing and increases by one the the value that indicates how many times a resource is simultaneously in use.

Function `CRMGetIndResource(theType: ResType; index: INTEGER): Handle;`

Function `CRMGet1IndResource(theType: ResType; index: INTEGER): Handle;`

CRMGetNamedResource and CRMGet1NamedResource

`CRMGetNamedResource` and `CRMGet1NamedResource` call `GetNamedResource` and `Get1NamedResource` respectively, and return a handle to the specified communications resource. The Communications Resource Manager then adds the handle to the list of resources that it is managing and increases by one the the value that indicates how many times a resource is simultaneously in use.

Function `CRMGetNamedResource(theType: ResType; name: STR255): Handle;`

Function `CRMGet1NamedResource(theType: ResType; name: STR255): Handle;`

CRMGetIndex

`CRMGetIndex` returns the number of times a resource is simultameously in use for the specified handle. `CRMGetIndex` returns 0 if it does not find the handle in the Communications Resource Manager's linked-list.

Function `CRMGetIndex(theHandle: Handle): LONGINT;`

CRMReleaseResource

`CRMReleaseResource` releases the resource that is specified by `theHandle`.

Procedure `CRMReleaseResource(theHandle: Handle);`

▲ **Warning** Your application must only release communications resources by calling `CRMReleaseResource`. If it tries to release resources itself, you might not like the consequences.

CRMGetIndToolName

`CRMGetIndToolName` returns the name of a tool in `toolName`.

Function

```
CRMGetIndToolName(bundleType : OSType; index : INTEGER;
VAR toolName : STR63) : OSErr;
```

Description

`bundleType` specifies the type of tool: `ClassCM` (for connection tools), `ClassTM` (for terminal tools), or `ClassFT`(for file transfer tools).

`index` specifies which occurence of a particular type of tool to return. For example, if `index` =2, the Communications Resource Manager will return the second occurence of a particular class of tool that has been registered with it.

Resource mapping routines

All resources used by a tool should be referenced by a local ID, which the operating system (using the tool bundle resource) maps into the appropriate physical ID. Special K contains two functions that will help you keep things straight: to map from physical ID to local ID, use `CRMRealToLocalID`; to map from local ID to physical ID, use `CRMocalToRealID`.

CRMLocalToRealID

`CRMLocalToRealID` maps a local resource ID to a physical resource.

Function

```
CRMLocalToRealID(bundleType: ResType; toolID: INTEGER;
theKind: ResType; localID: INTEGER): INTEGER;
```

Description

`bundleType` specifies the type of tool (connection, file transfer, or terminal) for which the mapping is to take place.

`toolID` specifies the bundle resource for the tool.

The appropriate values for `bundleType` are:

```
CONST
    ClassCM      =      'cbnd';
    ClassFT      =      'fbnd';
    ClassTM      =      'tbnd';
```

CRMRealToLocalID

`CRMRealToLocalID` maps a physical resource ID to a local resource ID.

Function

```
CRMRealToLocalID(bundleType: ResType; toolID: INTEGER;
theKind: ResType; realID: INTEGER): INTEGER;
```

Description

`bundleType` specifies the type of tool (connection, file transfer, or terminal) for which the mapping is to take place.

The format for a connection tool bundle resource is shown next (in `Rez` format). The same resource type declaration holds for terminal tools and file transfer tools. More information about the bundle resources can be found in the individual chapters on the Connection Manager, the Terminal Manager, and the File Transfer Manager.

```
type 'cbnd' {                                   /* or tbnd, or fbnd */
        integer = $$CountOf(TypeArray) - 1;
        array TypeArray {
                literal longint;     /* Type */
                integer = $$CountOf(IDArray) - 1;
                wide array IDArray {
                        integer;     /* Local ID */
                        integer;     /* Actual ID */
                };
        };
};
```

Guidelines for how to register a device

Here are some basic guidelines for writing drivers, which are similar to the standard serial port drivers:

private storage — All private data storage can be referenced off of the `dCtlStorage` field of the `DCtlEntry` for the drivers involved.

low memory — Do not use any.

driver naming — Use driver names that will be guaranteed to be unique. For example, do not use .CIn/.COut.

driver csCodes — Support all of the `csCode` calls supported by the standard serial drivers. If you need additional `csCodes`, contact Developer Technical Support to reserve `csCodes`. `csCodes` below 256 are reserved for Apple Computer, Inc.

driver arbitration — Your drivers should not allow multiple `drvrOpen` calls to succeed.

Data structures

Each device in the Communications Resource Manager's linked-list has a `CRMRec` associated with it. For the `crmDeviceType` field, Apple Computer, Inc. has defined the following value for serial port devices:

```
CONST
      crmSerialDevice            =      1;
```

When adding a `CRMRec` to the Communications Resource Manager's linked-list with the `CRMInstall` routine, pass 0 for the `crmDeviceID` field. The device identifier will be assigned by the Communications Resource Manager.

The `crmAttributes` field in the `CRMRec` points to a serial port device-specific data structure:

```
TYPE
      CRMSerialPtr                =      ^CRMSerialRecord;
      CRMSerialRecord             =      RECORD
            version               :      INTEGER;

            inputDriverName       :      StringPtr;
            outputDriverName      :      StringPtr;
            name                  :      StringPtr;
            deviceIcon            :      Handle;

            ratedSpeed            :      LONGINT;
            maxSpeed              :      LONGINT;

            reserved              :      LONGINT;
      END;
```

version

`version` is the version of the `CRMSerialRecord` data structure. This is version 0 of `CRMSerialRecord`.

inputDriverName

`inputDriverName` is a pointer to a null-terminated string, which is the name of the input driver for the given serial port. This driver should behave like the standard input serial port drivers (`.AIn` and `.BIn`) and support the same `csCode` calls as the standard drivers.

outputDriverName

`outputDriverName` is a pointer to a null-terminated string, which is the name of the output driver for the given serial port. This driver should behave like the standard output serial port drivers (`.AOut` and `.BOut`) and support the same `csCode` calls as the standard drivers.

name

`name` is a pointer to a null-terminated string, which is the name associated with a given port.

deviceIcon

`deviceIcon` is a handle to a relocatable block that contains an icon and a mask associated with the given port. Pass `NIL` if no icon is available.

ratedSpeed

`ratedSpeed` is the maximum recommended speed in bits per second.

maxSpeed

`maxSpeed` is the maximum speed in bits per second that the hardware is capable of.

Searching for serial port devices

The following routine will search the Communications Resource Manager's linked-list for devices of a given type.

```
PROCEDURE FindSerialPorts;
VAR
    theCRM  :    CRMRecPtr;
    theCRMRec   :    CRMRec;
    theErr : CRMErr;
    theSerial : CRMSerialPtr;

    old : INTEGER;

BEGIN
    theErr := 0;                                    { error status }
    old := 0;                                       { index number of ports }
    WHILE (theErr = noErr) DO
    BEGIN
        WITH theCRMRec DO
        BEGIN
            crmDeviceType := crmSerialDevice;       { search for port with index
                                                        number greater than "old"}
            crmDeviceID := old;                     { to be filled in later }
        END;
        theCRM := @theCRMRec;
         theCRM := CRMRecPtr(CRMSearch(QElemPtr(theCRM)));

        IF theCRM <> NIL THEN                       { got one! }
        BEGIN
             theSerial := CRMSerialPtr(theCRM^.crmAttributes);
            old := theCRM^.crmDeviceID;

            WITH theSerial^ DO
            BEGIN
            END;
        END
        ELSE
        BEGIN
            theErr := 1;
        END;
    END; {while}
END;
```

Summary

Communications Resource Manager routines	*see page*
`CRMGet1IndResource(theType: ResType; index: INTEGER): Handle;`	nn
`CRMGet1NamedResource(theType: ResType; name: STR255): Handle;`	nn
`CRMGet1Resource(theType: ResType; theID: INTEGER): Handle;`	nn
`CRMGetCRMVersion: INTEGER;`	nn
`CRMGetHeader: QHdrPtr;`	nn
`CRMGetIndex(theHandle: Handle): LONGINT;`	nn
`CRMGetIndResource(theType: ResType; index: INTEGER): Handle;`	nn
`CRMGetIndToolName(bundleType : OSType; index : INTEGER; VAR toolName : STR63) : OSErr;`	nn
`CRMGetNamedResource(theType: ResType; name: STR255): Handle;`	nn
`CRMGetResource(theType: ResType; theID: INTEGER): Handle;`	nn
`CRMInstall(crmReqPtr: QElemPtr);`	nn
`CRMReleaseResource(theHandle: Handle);`	nn
`CRMRemove(crmReqPtr: QElemPtr): OSErr;`	nn
`CRMSearch(crmReqPtr: QElemPtr): QElemPtr;`	nn
`InitCRM:CRMErr;`	nn
`LocalToRealID(theClass: INTEGER; toolID: INTEGER; theKind: ResType; localID: INTEGER): INTEGER;`	nn
`RealToLocalID(theClass: INTEGER; toolID: INTEGER; theKind: ResType; realID: INTEGER): INTEGER;`	nn

Constants

```
CONST
{       Communications Resource Manager's linked-list type    }
        crmType             =       9;

{       Version of CRMRec data structure  }
        crmRecVersion       =       1;

{       local/real resource ID mapping    }
        ClassCM       =       'cbnd';
        ClassTM       =       'tbnd';
        ClassFT       =       'fbnd';
```

Data types

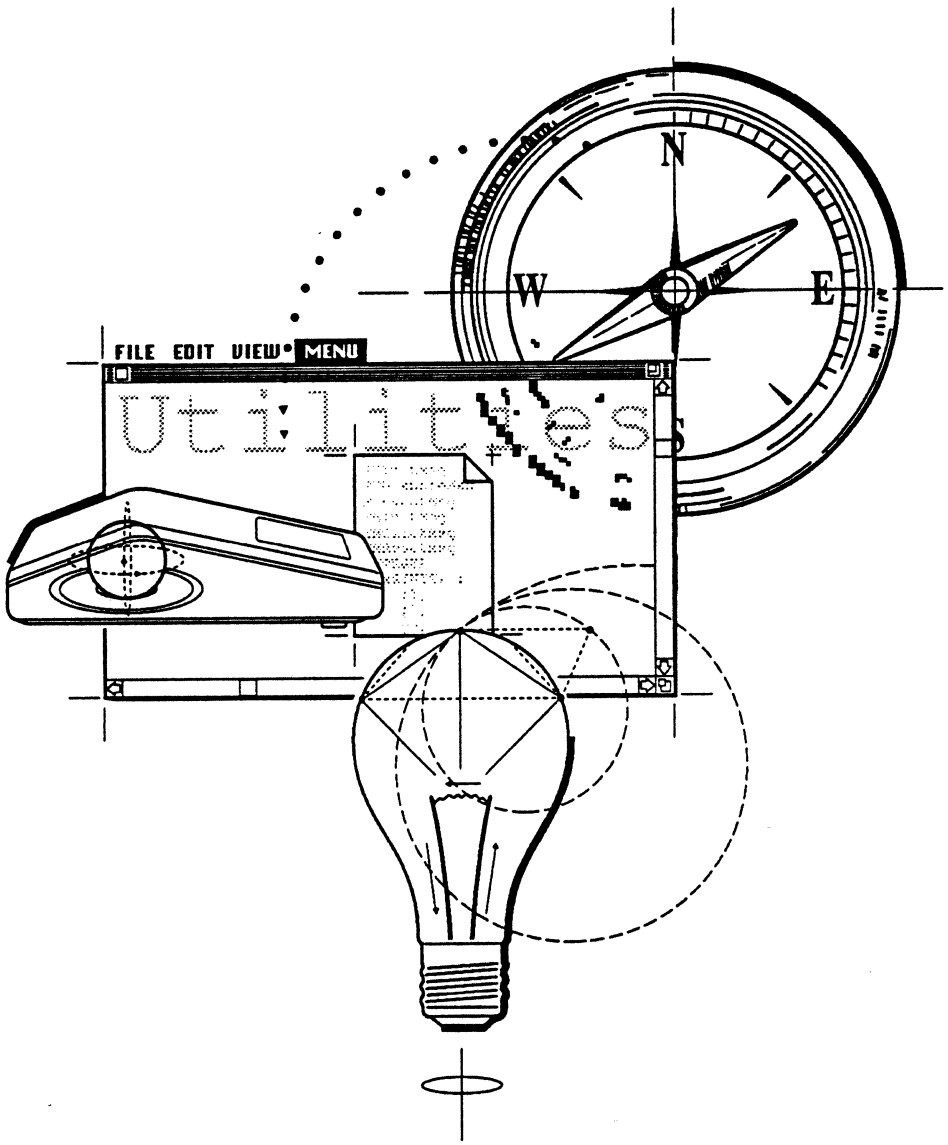
```
TYPE
        CRMRecPtr                   =       ^CRMRec;
        CRMRec                      =       RECORD
                qLink               :       QElemPtr;
                qType               :       INTEGER;
                crmVersion          :       INTEGER;

                crmPrivate          :       LONGINT;
                crmReserved         :       INTEGER;

                crmDeviceType       :       LONGINT;
                crmDeviceID         :       LONGINT;
                crmAttributes       :       LONGINT;
                crmStatus           :       LONGINT;

                crmRefCon           :       LONGINT;
        END;
```

Communications Resource Manager routine selectors

```
CRMGet1IndResource  .EQU   1290         CRMGet1NamedResource.EQU   1292

CRMGet1Resource     .EQU   1288         CRMGetCRMVersion   .EQU    1286

CRMGetHeader        .EQU   1282         CRMGetIndex        .EQU    1294

CRMGetIndResource   .EQU   1289         CRMGetIndToolName  .EQU    1297

CRMGetNamedResource .EQU1291            CRMGetResource     .EQU    1287

CRMInstall          .EQU   1283         CRMLocalToRealID   .EQU    1295

CRMRealToLocalID    .EQU   1296         CRMReleaseResource .EQU    1293

CRMRemove           .EQU   1284         CRMSearch          .EQU    1285
```

```
InitCRM             .EQU   1281
```

Chapter 7 **Utilities**



About this chapter and the utilities

This chapter describes the Special K utilities, which are a set of routines that make it easier for you to write applications. The Special K utilities are intended to be used for three specific tasks: manipulating dialog item lists, controlling pop-up menus, and searching a network for AppleTalk entities. Also described in this chapter are two routines your application can use to initialize the utilities and obtain the version number of the utilities.

Of the seven routines, two of them (`NuLookup` and `NuPLookup`) can be used only for networking or communications programming; the others can also be used in stand-alone applications.

This chapter contains a description of the routines and data structures that are the Special K utilities. At the end of the chapter is a summary that provides a quick reference to routines and data structures.

To use the dialog item list manipulation routines, you need to be familiar with:

- Dialog Manager (see *Inside Macintosh*, Volumes: IV, V)
- Control Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Resource Manager (see *Inside Macintosh*, Volumes: I, IV, V)

To use the network look-up utilities, you need to be familiar with:

- AppleTalk (see *Inside Macintosh*, Volumes: II, V)
- Custom Standard File dialog boxes (see *Inside Macintosh*, Volumes: I, IV)

Special K utilities

This section explains the routines and data structures that are the Special K utilities.

InitCTBUtilities Initializing Special K

InitCTBUtilities initializes the Special K utilities. Your application should call this routine after calling the standard Macintosh Toolbox initialization routines and before calling other Special K initialization routines, with one exception: If your application uses the Communications Resource Manager, it must initialize that before the Special K utilities.

Your application should check for the presence of the Special K before calling this function.

Function

```
InitCTBUtilities: CTBErr;
```

CTBGetCTBVersion Getting the Special K version number

CTBGetCTBVersion returns the version of the Special K utilities.

Function

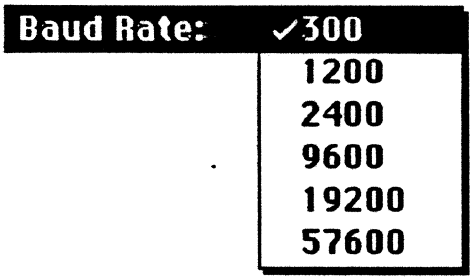
```
CTBGetCTBVersion: INTEGER;
```

NewControl Enhanced pop-up menu control

Special K includes a control definition procedure (CDEF) that extends the function of PopUpMenuSelect, which is a part of the Menu Manager in the Macintosh Toolbox. This CDEF, with resource ID=2, is available on Maintosh computers running System 4.2 or later.

Your application creates a pop-up menu control the same way as any other Macintosh control. However, when your application specifies the Special K NewControl routine, some of the parameters have different meanings than the Macintosh Toolbox NewControl parameters. Figure 7-1 shows a pop-up menu control.

Figure 7-1 Pop-up menu control



Function

```
NewControl (theWindow: WindowPtr; boundsRect: Rect;
title:STR255; visible: BOOLEAN; value: INTEGER; min,max:
INTEGER; procID:. INTEGER; refCon: LONGINT): ControlHandle;
```

Description

`theWindow` is the window the new pop-up menu will belong to. All coordinates pertaining to the pop-up menu will be interpreted in this window's local coordinate system.

`boundsRect` is the rectangle that encloses the pop-up menu; it determines the size and location of the rectangle. Your application must specify `boundsRect` in the local coordinates of `theWindow`.

`title` is the title of the pop-up menu. Be sure the title will fit in the pop-up menu's enclosing rectangle; if it doesn't fit, it will be truncated on the right for check boxes and radio buttons, or centered and truncated on both ends for simple buttons. To create a pop-up menu that has no title, have your application pass an empty string in this field.

`visible` specifies whether or not `NewControl` will draw the pop-up menu.

`value` specifies the manner in which the text in the menu is to be justified and highlighted. Valid values for `value` are:

```
popupTitleBold       =$00000100;
popupTitleItalic     =$00000200;
popupTitleUnderline  =$00000400;
popupTitleOutline    =$00000800;
popupTitleShadow     =$00001000;
popupTitleCondense   =$00002000;
popupTitleExtend     =$00004000;
popupTitleNoStyle    =$00008000;
popupLeftJust        =$00000000;
popupCenterJust      =$00000001;
popupRightJust       =$000000FF;
```

To have `NewControl` draw the pop-up with more than one of the characteristics listed above, pass in `value` the sum of all required characteristics.

Once a pop-up menu has been created, `NewControl` will set `value` to its minimum valid value. Your application can reset `value` with `SetCtlValue`.

`min` represents the resource ID of the menu that will be displayed in the pop-up control.

`max` contains the width of the pop-up title area.

`procID` should be an integer equal to 16*2+the appropriate variation code. Variation codes are discussed under "About variation codes."

If `refCon` contains a value, `NewControl` typecasts it to `ResType`. `NewControl` then calls `AddResMenu` for the menu associated with the pop-up menu control. For example, if `refCon` is `LONGINT('FONT')`, the menu in the pop-up menu control contains the names of all of the fonts currently installed. If `refCon` is `NIL`, `NewControl` does not do this.

About variation codes

Your application specifies variation codes when it passes a value in `procID`. Variation codes alter the characteristics of pop-up menu controls. To specify the appropriate variation code, your application sums the values that correspond to the desired pop-up menu characteristics. Below are the valid values:

variation code value	description
1	Provides constant control width. If your application specifies this value, `NewControl` will not resize the control. The width of the pop-up box will equal the width of the control minus the width of the pop-up title your application specified when it created the control. If the contents of the pop-up box do not fit into the control, the title will be truncated to a size that does fit and the utility will append an ellipsis (...).
2	Uses color QuickDraw. If your application specifies this value, `NewControl` will use the colors stored in the menu color table (`mctb`) for the color of the pop-up box when Color QuickDraw is available. If Color QuickDraw is not available, the utility will draw the pop-up box in black and white. If the owning `grafPort` is an old-style (classic QuickDraw) `grafPort`, the pop-up menu control will attempt to create a `cGrafPort` to draw the pop-up menu control colors and then dispose of it when finished drawing. By using a `cGrafPort`, the control avoids converting Color QuickDraw colors to classic QuickDraw colors.
4	If your application specifies this value, the pop-up CDEF will take the `refCon` field and treat it as a `ResType` and perform an `AddResMenu` with this resource type on the menu. The `NewControl` routine gets `refCon` from your application. The `GetNewControl` routine gets `refCon` from the control template.

8 Varies the font and size in the control If your application specifies this value, the utility will draw the pop-up with the font and size of the owning `grafPort`. The pop-up menu, when "popped-up" will also use the font and size specified by the `grafPort` that owns the control, instead of using the standard system font.

After the pop-up has been created

After `NewControl` has created the pop-up menu, `min` contains 1, `max` contains the number of items in the menu that is associated with the control, and `refCon` becomes available for the application to use.

In the process of creating the new control, `NewControl` may modify `boundsRect` to reflect the actual width of the pop-up box.

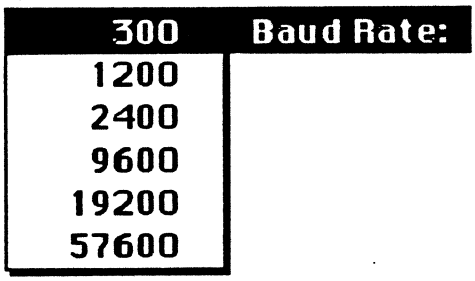
Other pop-up menu control characteristics

There are three pop-up menu control characteristics that you need to be familiar with : how the width of the control is resized, how the control changes in regards to system justification, and how your application can access the menu handle.

Whenever the pop-up control is redrawn, the utility recalculates the size of the menu associated with the control to handle the potential addition of new items in the menu by calling `CalcMenuSize`. `NewControl` also updates the width of the pop-up menu control to the sum of the width of the pop-up title, the width of the longest item in the menu (the menuWidth field of the menu information record), and some slop. As previously described, your application can override this characteristic by specifying variation code 1.

When the system justification is `teJustRight`, the pop-up control will be drawn similar to Figure 7-2.

Figure 7-2 Pop-up menu control when system justification is `teJustRight`.



Note that the positions of the pop-up box and the pop-up title are reversed from the standard positions show previously.

Your application can obtain the menu handle and the menu ID for the menu associated with the pop-up control by dereferencing the `contrlData` field of the control record. The `contrlData` field is a handle to a block of private information, the first four bytes of which is the menu handle, the next two bytes are the menu ID for the menu associated with the control.

Manipulating dialog item lists (DITLs)

As a logical extension to the Dialog Manager routines in the Macintosh Toolbox, Special K provides three procedures to append, shorten, and count the number of items in dialog item lists. These routines can be used regardless of whether or not you are programming a communications application or tool.

`AppendDITL` Appending to a dialog item list

`AppendDITL` lets your application append dialog items to an already exisiting dialog box.

Procedure

```
AppendDITL(theDialog: DialogPtr; theDITL: Handle; method:
INTEGER);
```

Description

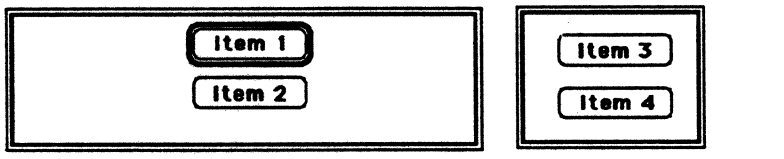
`theDialog` is a pointer to the dialog box into which you want to append an item list.

`theDITL` specifies the item list that you want to append. `method` specifies the manner in which you want the new item list appended: overlay, right, or bottom. Below are the acceptable vaules for `method`, followed by examples of the results of each method.

```
CONST
    overlayDITL       = 0;
    appendDITLRight   = 1;
    appendDITLBottom  = 2;
```

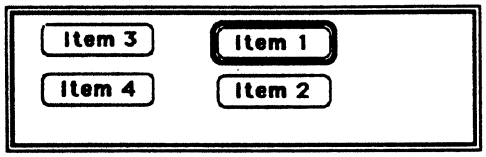
Figure 7-3 shows the initial dialog box, containing item 1, item 2, and the items to be appended, namely item 3 and item 4.

Figure 7-3 Initial dialog box and to-be-appended items



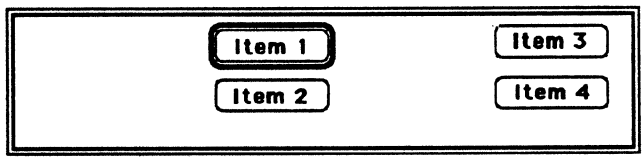
If your application uses `overlayDITL`, `AppendDITL` overlays the items in the to-be-appended dialog item list onto the dialog item list associated with `theDialog`, as shown in Figure 7-4.

Figure 7-4 Dialog box after appended items are overlaid.



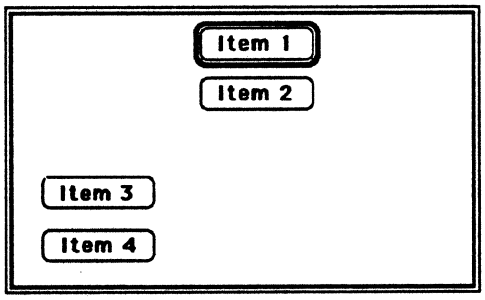
If your application uses `appendDITLRight`, `AppendDITL` offsets the items in the to-be-appended dialog item list by the top/right coordinates of `theDialog^.portRect`, as shown in Figure 7-5. Then `AppendDITL` appends the list to the end of the dialog item list associated with `theDialog`. `AppendDITL` automatically expands the dialog box as needed.

Figure 7-5 Dialog box after items appended to the right.



If your application uses `appendDITLBottom`, `AppendDITL` offsets the items in the to-be-appended dialog item list by the bottom/left coordinates of `theDialog^.portRect`, as shown in Figure 7-6. Then `AppendDITL` appends the list to the end of the dialog item list associated with `theDialog`, and expands the dialog box as needed.

Figure 7-6 Dialog box after items appended to the bottom.



If you know your application will need to restore a window to its size before an `AppendDITL`, your application should save that size before it calls `AppendDITL`. `ShortenDITL`, which is the procedure that shortens dialog item lists (and is described on page nn) will not automatically resize the dialog box.

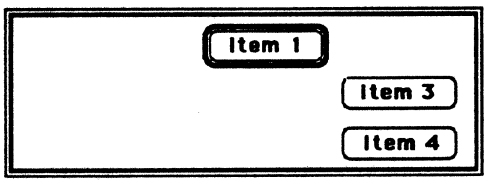
Because `AppendDITl` modifies the contents of `theDITL`, your application can get rid of `theDITL` after calling `AppendDITL`. A typical calling sequence is :

```
theDITL := GetResource('DITL', theID);
AppendDITL(theDialog, theDITL, appendDITLBottom);
ReleaseResource(theDITL);
```

Special ways to append items

Your application can append a new dialog item list relative to the location of specific items in the dialog box, rather then appending new dialog items relative to the coordinates of `Dialog^.portRect`. To do this, your application uses a negative number in the `method` parameter, where the number corresponds to the item that is to be the point of reference. For instance, if `method` is –2, then the items in the to-be-appended dialog item list will have their item boxes offset by the `topLeft` of the item box for item 2 in `theDialog`. Figure 7-7 shows how item 3 and item 4 were appended relative to the position of item 2.

Figure 7-7 DITL displayed after an append relative to item 2.



`CountDITL` Counting the number of items in a list

`CountDITL` returns the number of items in the dialog item list that is associated with `theDialog`.

Function

```
CountDITL(theDialog:DialogPtr): INTEGER;
```

`ShortenDITL` Shortening a dialog item list

`ShortenDITL` removes items from the end of the given dialog item list, but does not automatically resize the dialog box. If you know that your application will need to resize the dialog box, save this size prior to calling `AppendDITL`.

Procedure

```
ShortenDITL(theDialog: DialogPtr; numberItems: INTEGER);
```

Description

`theDialog` specifies the dialog box to be shortened.

`numberItems` specifies the number of items to be removed.

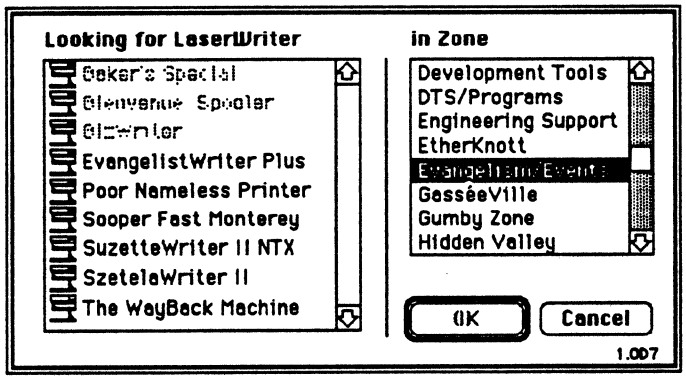
Showing AppleTalk entities: NuLookUp and NuPLookUp

The network look-up utilities, `NuLookup` and `NuPLookup`, allow your application to present the user with a dialog box that contains AppleTalk entities. By providing either `NuLookup` or `NuPLookup` with the proper parameters, your application can include in the dialog box one or more different types of AppleTalk entities. Both `NuLookup` and `NuPLookup` perform much the same task, but `NuPLookup` gives the programmer a bit more flexibility.

The Special K utilities also include filter and hook routines that your application can take advantage of to customize the dialog box or to filter out information that would otherwise be included in it. These routines are described below under "Customizing the dialog box with hook and filter procedures.".

The results of `NuLookup` and `NuPLookup` are displayed in a dialog box similar to Figure 7-8, which shows the results of a search for LaserWriter printers in the zone "Evangelism/Events".

- **Figure 7-8** Network look-up dialog box



`NuLookp` Network lookups

`NuLookp` returns to your application the name/object/zone tuple and AppleTalk node/network/zone numbers tuple for the item that the user selected..

When your application first calls `NuLookup`, a zone list is built (if possible). Then `NuLookup` makes a synchronous NBP lookup for the specified objects. Next, the preliminary object list is built, and the dialog box is presented to the user. At all times while the dialog box is displayed, an asynchronous NBP lookup with long retry and timeout is kept going. Objects in the name list are aged, so that if an object misses several consecutive asynchronous NBP lookups, it is removed from the list.

Both the zone and name lists are alphabetized using the international utilities. Since they use the standard Macintosh application font, they support the different script systems for Macintosh.

Function

```
NuLookup(where: Point; prompt: STR255; numTypes: INTEGER;
typeList: NLType; nameFilter: ProcPtr; zoneFilter:
ProcPtr; hookProc: ProcPtr; VAR theReply: LookupReply):
INTEGER;
```

`where` indicates in global coordinates where `NuLookup` should place the top left corner of the look-up dialog box. `prompt` is a string that is displayed at the top of the look-up dialog box. In Figure 7-8, the string "Looking for LaserWriter" was passed to `NuLookup`.

`numTypes` is the number of object types that will be included in the lookup.

`typeList` is a structure of type `NLType`, which is an array of AppleTalk object types, along with a handle to an icon. If no icon is required, pass NIL for `theIcon`.

```
TYPE
    NLType              =     ARRAY[0..3] OF RECORD
          theIcon       :     Handle;
          typeStr       :     Str32;
    END;
```

◆ *Assembly Note.* From assembly language, more than four object types may be specified by passing a pointer to an array with the required number of items.

`nameFilter` is a pointer to a procedure that will filter out name/object/zone tuples from the network look-up dialog box. `zoneFilter` is a pointer to a procedure that will filter out zones from the network look-up dialog box. `hookProc` is a pointer to a hook procedure that can be used to modify the behavior of items in the dialog box or to call a background procedure. These three procedures are described "Customizing the dialog with hook and filter procedures" on page nn. If you do not need these routines in your application, specify `NIL`.

`theReply` is the look-up reply record that contains the name/object/zone tuple for the object, if any, that was selected by the user, as well as the AppleTalk address consisting of node/network/zone numbers.

```
    LookupReply             =     RECORD
          theEntity     :     EntityName;
          theAddr       :     AddrBlock;
    END;
```

When your application is initially passes the `theReply` data structure into the `NuLookup` procedure, `theReply.theEntity` should contain the default zone and name. If the specified object is not in the list of accepted objects in `typeList`, the specified object is ignored, and only the default zone is set. If an appropriate match is found in the initial look up, when the dialog box comes up, the specified zone will be selected, as well as the specified name of the given object.

Pressing the Return key acts the same as pressing the OK button. Cancel is selected by holding down the Command key and pressing the Period key. The Up Arrow key and the Down Arrow key change the selected name to either the cell above or the cell below. Holding down the Command key while pressing the Up Arrow key or the Down Arrow key will change the selected zone up or down one cell.

`NuLookup` will return one of three values:

```
CONST
    nlOk        =    0;
    nlCancel    =    1;
    nlEject     =    2;
```

`nlOk` is returned when the user selects the "OK" button in the dialog box. `nlCancel` is returned if the user selects the "Cancel" button. `nlEject` is returned if the dialog box is aborted somehow through use of the hook procedure.

`NuPLookup` A more versatile network lookup

`NuPLookup` performs much the same task as `NuLookup`, except that it gives the programmer even greater control over customization of the `NuLookup` dialog box. Additional parameters that can be specified are `userData`, `dialogID`, and `filterProc`.

Function

```
NuPLookup(where: Point; prompt: STR255; numTypes:
INTEGER; typeList: NLType; nameFilter: ProcPtr;
zoneFilter: ProcPtr; hookProc: ProcPtr; userData:
LONGINT; dialogID: INTEGER; filterProc: ProcPtr; VAR
theReply: LookupReply): INTEGER;
```

`userData` is a field that the user can specify. It may be referenced from the hook procedure or the filer procedure with the

`refCon` field of the dialog box record. `refCon` is a handle to the `userData` value.

The following code fragment demonstrates how to access the `userData` field:

```
TYPE
    LongH           =    ^LongPtr;
    LongPtr     =    ^LONGINT;
```

```
BEGIN
    myUserData :=    LongH (GetWRefCon (theDialog))^^;
END;
```

`dialogID` is the resource ID for a dialog box (and for the corresponding dialog item list) that is to replace the standard look-up dialog box. All of the items in the replacement dialog item list must correspond to items in the standard dialog item list, although they can be moved around. The list of standard items and their placement is shown below:

Item number	Type	Rectangle (top, left, bottom, right)
1	OK button	{175, 240, 195, 310}
2	Cancel button	{175, 320, 195, 390}
3	Default hilite (userItem)	{175, 240, 195, 310}
4	Title (staticText)	{5, 15, 19, 226}
5	Item list (userItem)	{25, 15, 192, 210}
6	Zone list title (staticText)	{5, 240, 19, 391}
7	Zone list (userItem)	{25, 240, 147, 391}
8	Line (userItem)	{25, 225, 196, 226}
9	Version (userItem)	{200, 360, 210, 400}
10-13	Reserved	

`filterProc` is a standard dialog box filter procedure that is called after the standard `NuLookup` modal dialog box filter procedure. The format of the filter procedure is the same as a standard modal dialog box filter procedure:

Customizing the dialog box with hook and filter procedures

You can customize the operation of the dialog box for specific applications by using the filter procedures and the hook procedure. Filter procedures are used to filter out zones for inclusion in the zone list or to filter out objects from the object list. The hook procedure is used to modify the behavior of items in the dialog box and can also be used to call a background procedure.

`MyNameFilter` Name filters

Before each item name is included in the network look-up dialog list, it is passed to the name filter procedure for processing. Specify `NIL` for no filter procedure.

Function `MyNameFilter(theEntity: EntityName): INTEGER;`

This filter procedure is passed the network entity in `theEntity` and will return an integer with one of the following values:

```
CONST
    nameInclude     =    1;
    nameDisable     =    2;
    nameReject =     3;
```

`nameInclude` results in the `theEntity` being included in the name list of the network look-up dialog box. `nameDisable` includes `theEntity` but disables it; the item in the list will be visible (although dimmed), but not selectable. `nameReject` causes `theEntity` not to appear in the lists.

`MyZoneFilter` Zone filters

Before each zone item is included in the network look-up dialog list, it is passed to the zone filter procedure for processing. Specify `NIL` for no filter procedure. The format for the filter is shown below:

Function `MyZoneFilter(theZone: STR32): INTEGER;`

This filter procedure is passed the AppleTalk zone in `theZone` and returns an integer with the following values:

```
CONST
    zoneInclude     =    1;
    zoneDisable     =    2;
    zoneReject =     3;
```

`zoneInclude` results in the `theZone` being included in the zone list in the network look-up dialog. `zoneDisable` includes `theZone` but disables it; the item in the zone list will be visible (although dimmed), but not selectable. `zoneReject` causes `theZone` not to appear in the zone list.

`MyHookProcedure` The hook procedure

The hook procedure is called by `NuLookup` immediately after `ModalDialog` is called and before the standard hook procedure is called. `ModalDialog` returns a number that corresponds to the item hit in the dialog box. `NuLookup` employs a modal dialog box filter procedure that returns the item number for physical items hit in the dialog box, as well as the item numbers that correspond to virtual items.

Function `MyHookProc(item: INTEGER; theDialog: DialogPtr): INTEGER;`

Appropriate virtual and real dialog items are:

```
CONST
    hookOK              =     1;
    hookCancel          =     2;
    hookOutline         =     3;
    hookTitle           =     4;
    hookItemList        =     5;
    hookZoneTitle       =     6;
    hookZoneList        =     7;
    hookLine            =     8;
    hookVersion         =     9;
    hookReserved1       =     10;
    hookReserved2       =     11;
    hookReserved3       =     12;
    hookReserved4       =     13;

{   virtual items in dialog item list      }
    hookNull            =     100;
    hookItemRefresh     =     101;
    hookZoneRefresh     =     102;

    hookEject           =     103;

    hookPreflight       =     104;
    hookPostflight      =     105;

    hookKeyBase         =     1000;
```

The first thirteen items correspond to physical items in the dialog item list. The other items are "virtual" items that correspond to certain actions that may need to be performed.

`hookNull` is a fake event that corresponds to a null event. The standard modal dialog box filter procedure returns `hookNull` in `itemHit` for null events.

`hookItemRefresh` causes the item list in the look-up dialog box to be discarded and regenerated.

`hookZoneRefresh` causes the zone list in the look-up dialog box to be discarded and regenerated. This will also cause a `hookItemRefresh` event to be generated afterwards, as a side effect.

`hookEject` causes all outstanding NBP lookups to be terminated and `NLeject` to be returned by `NuLookup`.

`hookPreflight` is processed after the zone and object lists are formed, but before the dialog box is displayed.

`hookPostflight` is processed before the dialog box is disposed of.

Items greater than `hookKeyBase` are actually the ASCII value of the key that is pressed, offset by `hookKeyBase`. For example, an `itemHit` of 1032 decimal would correspond to a `keyDown` event with the character generated being a space (ASCII 32 decimal).

Summary

Utility routine	*see page*
`InitCTBUtilities;`	nn
`AppendDITL(theDialog: DialogPtr; theDITL: Handle; method: INTEGER);`	nn
`CountDITL(theDialog: DialogPtr): INTEGER;`	nn
`ShortenDITL(theDialog: DialogPtr; numberItems: INTEGER);`	nn
`NuLookup(where: Point; prompt: STR255; numTypes: INTEGER; typeList: NLType; nameFilter: ProcPtr; zoneFilter: ProcPtr; hookProc: ProcPtr; VAR theReply: LookupReply): INTEGER;`	nn
`NuPLookup(where: Point; prompt: STR255; numTypes: INTEGER; typeList: NLType; nameFilter: ProcPtr; zoneFilter: ProcPtr; hookProc: ProcPtr; userData: LONGINT; dialogID: INTEGER; filterProc: ProcPtr; VAR theReply: LookupReply): INTEGER;`	nn

Routines in your applicatin	*see page*
`MyNameFilter(theEntity: EntityName): INTEGER`	nn
`MyZoneFilter(theZone: STR32): INTEGER;`	nn
`MyHookProc(item: INTEGER; theDialog: DialogPtr): INTEGER;`	nn

Data structures

```
TYPE
      NLType              =     ARRAY[0..3] OF RECORD
            theIcon       :     Handle;
            typeStr       :     Str32;
      END

      LookupReply         =     RECORD
            theEntity     :     EntityName;
            theAddr       :     AddrBlock;
      END;
```

Constants

```
CONST
        nlk             =       0;
        nlCancel        =       1;
        nlEject         =       2;


{       values that name filterProc returns }
        nameInclude             =       1;
        nameDisable             =       2;
        nameReject              =       3;

{       values that zone filterProc returns }
        zoneInclude             =       1;
        zoneDisable             =       2;
        zoneReject              =       3;

{       dialog items for hook procedure  }
        hookOK                  =       1;
        hookCancel              =       2;
        hookOutline             =       3;
        hookTitle               =       4;
        hookItemList            =       5;
        hookZoneTitle           =       6;
        hookZoneList            =       7;
        hookLine                =       8;
        hookVersion             =       9;
        hookReserved1           =       10;
        hookReserved2           =       11;
        hookReserved3           =       12;
        hookReserved4           =       13;

{       virtual items in dialog item list       }
        hookNull                =       100;
        hookItemRefresh         =       101;
        hookZoneRefresh         =       102;

        hookEject               =       103;

        hookPreflight           =       104;
        hookPostflight          =       105;


        hookKeyBase             =       1000;
```

Standard item placement

Item number	Type	Rectangle (top, left, bottom, right)
1	OK button	{175, 240, 195, 310}

2	Cancel button	{175, 320, 195, 390}
3	Default hilite (userItem)	{175, 240, 195, 310}
4	Title (staticText)	{5, 15, 19, 226}
5	Item list (userItem)	{25, 15, 192, 210}
6	Zone list title (staticText)	{5, 240, 19, 391}
7	Zone list (userItem)	{25, 240, 147, 391}
8	Line (userItem)	{25, 225, 196, 226}
9	Version (userItem)	{200, 360, 210, 400}
10-13	Reserved	

Special resources

Pop-up menu control

Parameter	Before NewControl	After NewControl
min	ID of menu to use	1
max	width of pop-up title	number of menu items
value	currently selected item	currently selected item
refCon	resource type to append to menu using `AddResMenu`	available to application

Constants

```
CONST
{       DITL manipulation constants       }
        overlayDITL             =       0;
        appendDITLRight         =       1;
        appendDITLBottom        =       2;
```

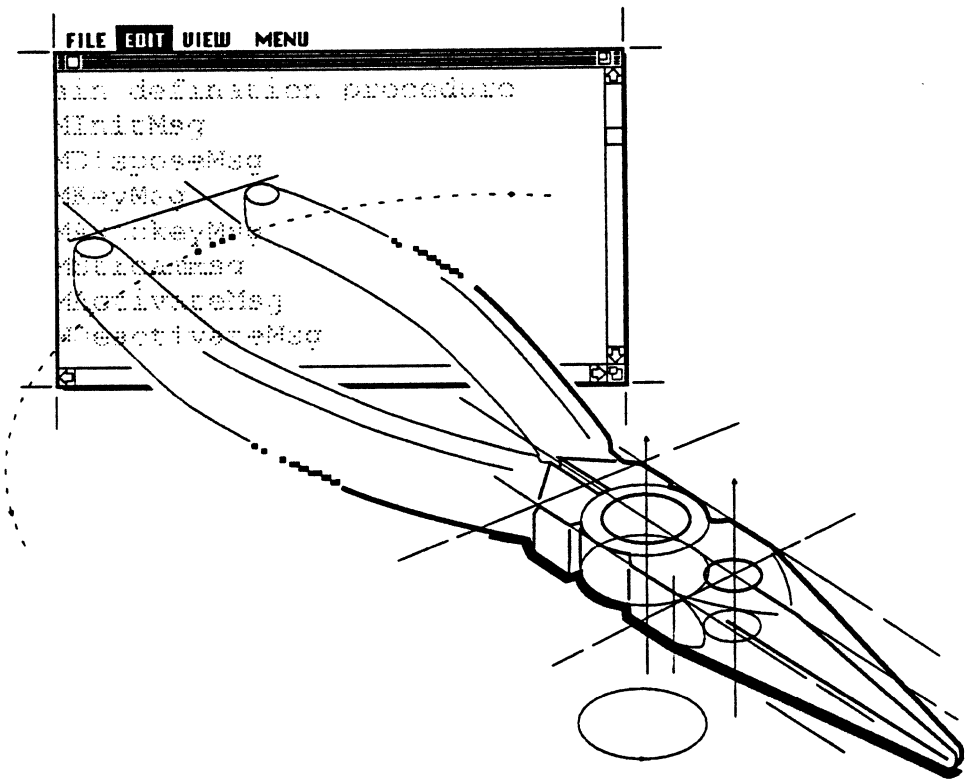
Resource formats

```
type 'cbnd' {                                   /* or tbnd, or fbnd */
        integer = $$CountOf(TypeArray) - 1;
        array TypeArray {
                literal longint;        /* Type */
                integer = $$CountOf(IDArray) - 1;
                wide array IDArray {
                        integer;        /* Local ID */
                        integer;        /* Actual ID */
                };
        };
};
```

Utility routine selectors

```
InitCTBUtilities    .EQU    1025        AppendDITL          .EQU    1026

CountDITL           .EQU    1027        ShortenDITL         .EQU    1028

CTBGetCTBVersion    .EQU    1029        NuLookup            .EQU    1030

NuPLookup           .EQU    1031
```

Chapter 8 **Fundamentals of Writing Your Own Tool**



About this chapter

This chapter provides general information about writing a connection, file transfer, or terminal emulation tool. You can find more specific information on writing these tools in Chapter 9, "Writing Connection Tools," Chapter 10, "Writing Terminal Tools," and Chapter 11, "Writing File Transfer Tools." You need to read at least two chapters to learn how to write a tool: this chapter, and the chapter that deals specifically with your type of tool. The information in this chapter is common to all three types of tools.

This chapter starts off with high-level information about writing a tool. Then, it describes the six code resources that are an essential part of any communications tool meant to be used with Special K. After that, the chapter provides example code to give you a better idea of what you need to do to write a tool.

To write your own communications tool, you need to be familiar with the chapter in this book that discusses the manager with which your tool will interface, either Chapter 3, "Connection Manager," Chapter 5, "Terminal Manager," or Chapter 6 "File Transfer Manager." You should also be familiar with the Apple guidelines for communications tools, which is discussed in Appendix A, "Guidelines for Communications Tools."

You need to know the following topics:

- Serial Driver (see *Inside Macintosh*, Volumes: II, IV)
- AppleTalk (see *Inside AppleTalk* and *Inside Macintosh*, Volumes: II, IV, V)
- Dialog Manager (see *Inside Macintosh*, Volumes: I, IV, V)
- Script Manager (see *Inside Macintosh*, Volume V)
- Creating stand-alone code (see Apple Technical Note 110)

About writing a tool

The Communication Toolbox managers interact with an application in the same way that the Macintosh Toolbox managers interact with an application; the application calls a routine, which the appropriate manager handles by sending a message off to a tool. For example, when an application requires a connection service such as reading data from a remote entity, it calls the `CMRead` routine. The Connection Manager takes this request and passes it on by issuing a message, `cmReadMsg`, to the main code resource of the appropriate tool.

Most of the messages that the Special K managers send out are similar to the messages that the other Special K managers send out. This is because all of the managers have to handle similar tasks, like tool selection, record validation, and string localization. Notice the similarity of messages in a request to perform tool-selection: the Connection Manager sends a `CMChoose`, the Terminal Manager sends a `TMChoose`, and the File Transfer Manager sends a `FTChoose`.

Because the majority of messages in one of the managers are similar to those in the others, this chapter shows you how to handle only Connection Manager messages. While you may not be writing a connection tool, you can still learn from the sample code that shows how a connection tool handles messages from the Connection Manager, and apply this knowledge toward writing your own tool.

To find a functional description of the routine associated with a message, read the appropriate section earlier in this book.

The six essential resources

You need to create sex resources to make your own tool. Five of these are code resources and the other is a tool-related resources. All the resources must have the same resource ID for a given tool. Remember, the description below is for a connection tool. Resource descriptions for a terminal tool are provided in Chapter 10, "Writing Terminal Tools" and resource descriptions for a file transfer tool are provided in Chapter 11, "Writing File Transfer Tools."

There is one tool-related resource that you need to form a tool:

cbnd	bundle resource, which has the name of the tool and contains information on what resources "belong" to the tool. For terminal emulation tools this resource is called *tbnd*, and for file transfer tools, this resource is called *fbnd*.

There are five code resources that you need to form a tool:

cdef	main code resource that performs the basic communications functions, such as `CMNew`, `CMRead`, and `CMWrite`. This resource is discussed in Chapter 9, "Writing a Connection Tool." For terminal emulation tools this resource is called *tdef* and is discussed in Chapter 10, "Writing Terminal Tools" ; for file transfer tools this resource is called *fbnd* and is discussed in Chapter 11, "Writing File Transfer Tools."
cval	validation resource that validates connection records with `CMValidate` and also fills in configuration record default values with `CMDefault` For terminal emulation tools this resource is called *tval*, and for file transfer tools, this resource is called *fval*.
cset	setup resource that performs operations necessary to support putting up user interface elements that configure a connection record associated with a tool. For terminal emulation tools this resource is called *tset*, and for file transfer tools, this resource is called *fset*.
cscr	scripting interface resource that handles the interface between a scripting language and the tool. For terminal emulation tools this resource is called *tscr*, and for file transfer tools, this resource is called *fscr*.
cloc	localization resource that handles localizing configuration strings. For terminal emulation tools this resource is called *tloc*, and for file transfer tools, this resource is called *floc*.

The bundle resource

The connection bundle contains the master list of resources that are associated with your connection tool. Besides the six standard resources, the connection bundle should contain any additional resources that your tool requires, like dialogs or menus. The connection bundle should be a named resource, with the name of the resource being the name of the connection tool.

The Connection Manager accomodates multiple uses of the same tool at the same time by using the bundle resource. It does this by keeping track of the specifics of each instance of a tool in a connection record. However, because of this correspondence between a given instance of a tool and its associated connection record, be sure to index all resources when they are allocated and released so that the integrity of this one-to-one correspondence is maintained.

Also, your connection tool should refer to resources with local IDs that the Connection manager can map to actual resource IDs (the Communications Resource Manager `CRMLocalToRealID` and `CRMRealToLocalID` routines help you do this). The connection bundle resource, the format of which is shown next, must provide a data structure to accomodate this mapping.

```
type 'cbnd' {
       integer = $$CountOf(TypeArray) - 1;
       array TypeArray {
              literal longint;     /* Type                    */
              integer = $$CountOf(IDArray) - 1;
              wide array IDArray {
                     integer;      /* Local ID                */
                     integer;      /* Actual ID               */
              };
       };
};
```

The validation code resource

The purpose of the validation code resource is to parse two possible messages from the manager—in the case of the Connection Manager these are `cmValidateMsg` and `cmDefaultMsg`. An application or tool will request one of these services when it requires your tool to check the values in the connection record (for terminal tools this record is called the terminal record and for file transfer tools this record is called the file transfer record) or when it requires your tool to reset the connection record to its default values (your connection tool should contain the default values for the connection record).

The validation code resource, an example of which is below, should be a resource of type `cval` and be able to accept the parameters shown:

```
FUNCTION cval(hConn: ConnHandle; msg: INTEGER;
     p1, p2, p3: LONGINT): LONGINT;
```

```
VAR
    pConfig:    ConfigPtr;

BEGIN
    CASE msg OF
    cmValidateMsg:      { hConn is valid here }
        BEGIN
         cval = DoValidate(hConn);
        END;
    cmDefaultMsg:       { hConn is not valid here }
        BEGIN            { p1 is a pointer to the configPtr }
                         { p2 is allocate or not }
                         { p3 is the procID of the tool }
        IF p2 = 1 THEN
            BEGIN
              pConfig := ConfigPtr(NewPtr(SIZEOF(ConfigRecord)));
             ConfigHandle(p1)^ := pConfig;
             {   real programmers check errors here }
            END
        ELSE
            BEGIN
             pConfig := ConfigHandle(p1)^;
            END;
         DoDefault(pConfig);
        END;
    END; {case}
END;
```

The messages accepted by the validation code resource and their associated values are:

```
CONST
{       validation code resource messages }
        cmValidateMsg       =       0;
        cmDefaultMsg        =       1;
```

For each of the messages defined above, `p1`, `p2`, and `p3` take on different meanings, which are discussed under the description of each message.

cmValidateMsg

Your tool will receive `cmValidateMsg` when the application requires your tool to validate the fields in the connection record. Your tool should compare the values in this record with the values specified in your tool.

The sample code below shows how your tool can respond to a `cmValidateMsg`.

After executing the code necessary to respond to a `cmValidateMsg`, your code should pass back 0 if there were no errors, or 1 if the configuration record had to be rebuilt. `p1`, `p2`, and `p3` should be ignored.

```
{    perform validate here }
FUNCTION DoValidate(hConn: ConnHandle): LONGINT;
VAR
     pPrivate:    PrivatePtr;
     pConfig:     ConfigPtr;

BEGIN
     DoValidate := 0;          { optimism reigns }
      pConfig := ConfigPtr(hConn^^.config);
      pPrivate := PrivatePtr(hConn^^.private);

     IF pConfig^.foobar = 0 THEN
          DoValidate := 0      { okey dokey }
     ELSE
          DoValidate := 1;     { uh-oh }
END;
```

cmDefaultMsg

Your tool will receive `cmDefaultMsg` when the application requires your tool to fill in the fields of a connection record. Default values should be specified in your tool. The example code below shows how your tool can handle `cmDefaultMsg`.

After executing the code necessary to respond to `cmDefaultMsg`, `p1` should pass back a pointer to the configuration record pointer. `p2`, if it contained 1 when `CMDefault` was called, should allocate the configuration record and return the pointer in `p1`. If it was 0, then simply use the configuration pointer obtained by dereferencing `p1`.

```
PROCEDURE DoDefault(theConfig : ConfigPtr);

VAR    boo : STR255;

BEGIN
       WITH theConfig^ DO
              BEGIN
{      default is 9600 8 N 1   no handshaking                }

              baudrate := 9600;
              databits := data8;
              stopbits := stop10;
              paritybits := noParity;

              WITH theConfig^.shaker DO
              BEGIN
                     fXOn := 0;
                     fCTS := 0;
                     xOn := CHAR($11);
                     xOff := CHAR($13);
```

```
                errs := 0;
                evts := 0;
                fInX := 0;
                fDTR := 0;
            END;

            portName := GetFirstSerial;

            flags := 0;
            END;

END;

END.
```

The setup definition code resource

You might want your tool to present to users a custom dialog box that will allow them to configure their own connection or select a connection tool. The Connection Manager routines `CMSetupPreflight`, `CMSetupSetup`, `CMSetupItem`, `CMSetupFilter`, and `CMSetupCleanup` support this feature; your tool should be able to handle the messages associated with these routines.

The connection tool setup code resource should be a function called `cset` and be able to handle the following parameters:

```
{   main entry point for cset resource }
FUNCTION cset(pSetup: SetupPtr; msg: INTEGER;
    p1, p2, p3: LONGINT): LONGINT;
TYPE
    LocalHandle = ^LocalPtr;
    LocalPtr = ^LocalRecord;
    LocalRecord = RECORD                    { private tool setup
context }
        foobar: LONGINT;
    END;

    IntPtr = ^INTEGER;
    EventPtr = ^EventRecord;

BEGIN
    CASE msg OF
    cmSpreflightMsg:
        BEGIN
         theCookie := CookiePtr(NewPtr(SIZEOF(CookieRecord)));
         CookieHandle(p3)^ := theCookie;      { send back theCookie }
         cset := Preflight(pSetup, theCookie);
        END;
```

```
    cmSsetupMsg:
        BEGIN
         theCookie := CookieHandle(p3)^;      { get the magic cookie }
        Setup(pSetup);                        { do the setup }
        END;
    cmSitemMsg:
        BEGIN
         theCookie := CookieHandle(p3)^;        { get the magic cookie }
         Item(pSetup, theCookie, IntPtr(p1));  { process the items hit }
        END;
    cmSfilterMsg:
        BEGIN
         theCookie := CookieHandle(p3)^;      { get the magic cookie }
          cset := Filter(pSetup, theCookie, EventPtr(p1), IntPtr(p2));
        END;
    cmScleanupMsg:
        BEGIN
         theCookie := CookieHandle(p3)^;      { get the magic cookie }
         DisposPtr(Ptr(theCookie));           { and get rid of it }
        END;
    END; {case}
END;
```

Valid values for `msg` are:

```
CONST
       cmSpreflightMsg      =      0;
       cmSsetupMsg          =      1;
       cmSitemMsg           =      2;
       cmSfilterMsg         =      3;
       cmScleanupMsg        =      4;
```

For each of the messages defined above, `p1`, `p2`, and `p3` take on different meanings, which are discussed under the description of each message. When your tool handles these routines, it should use a data structure called `SetupStruct`.

```
TYPE
       SetupPtr             =      ^SetupStruct;
       SetupStruct          =      RECORD
             theDialog      :      DialogPtr;
             count          :      INTEGER;
             theConfig      :      Ptr;
             procID         :      INTEGER
       END;
```

cmSPreflightMsg

Your setup definition code resource should perform function similar to that shown below when it receives `cmSpreflightMsg` from the Connection Manager.

When passed to your connection tool, `p3` will be a pointer to a `LONGINT` that gets passed along to the other routines during setup definition. `p3` should serve as a magic cookie if the setup definition procedure requires some private context.

After executing the code necessary to respond to `cmSpreflightMsg`, your connection tool should return a handle to a dialog item list. This handle should then be disposed of by the caller of this function.

```
FUNCTION Preflight(pSetup: SetupPtr; theCookie: LocalPtr): LONGINT;
CONST
    localID = 1;                                { we want DITL local ID 1 }
VAR
    hDITL: Handle;
    theID: INTEGER;
    oldRF: INTEGER:

BEGIN
    theCookie^.foobar := 0;                     { setup theCookie }

     theID := CRMLocalToRealID(ClassCM, pSetup^.procID, 'DITL',
localID);
    IF theID = -1 THEN
        Preflight := 0                          { no DITL found }
    ELSE
    BEGIN
        oldRF := CurResFile;
        UseResFile(pSetup^.procID);         { procID is the tool refnum }
        hDITL := Get1Resource('DITL', theID);
        UseResFile(oldRF);

        IF hDITL <> NIL THEN
            DetachResource(hDITL);          { got it so detach it }

        Preflight := LONGINT(hDITL);
    END;
END;
```

cmSsetupMsg

Your setup definition code resource should perform function similar to that shown below when it receives `cmSsetupMsg` from the Connection Manager.

When passed to your connetion tool, `p3` will be a pointer to a `LONGINT` magic cookie value.

```
PROCEDURE Setup(pSetup: SetupPtr);
CONST
     myFirstItem = 1;
     mySecondItem = 2;

VAR
     first:  INTEGER;                    { first item appended (0 based) }
     pConfig:ConfigPtr;

BEGIN
     WITH pSetup^ DO
     BEGIN
          first := count - 1;                  { count is 1 based }
          pConfig := ConfigPtr(theConfig);     { get the config ptr }

          GetDItem(theDialog, first+myFirstItem, itemKind, itemHandle,
        itemRect);
          SetCtlValue(ControlHandle(itemHandle), pConfig^.foobar);

          GetDItem(theDialog, first+mySecondItem, itemKind, itemHandle,
        itemRect);
          SetCtlValue(ControlHandle(itemHandle), 1-pConfig^.foobar);

     END; {with}
END;
```

cmSitemMsg

Your setup definition code resource should perform function similar to that shown below when it receives `cmSitemMsg` from the Connection Manager.

When passed to your connection tool, `p1` will contain an item that was selected from the dialog item list and `p3` will contain a pointer to `magicCookie`.

```
PROCEDURE Item(pSetup: SetupPtr; pItem: IntPtr);
CONST
     myFirstItem = 1;
     mySecondItem = 2;

VAR
     first:  INTEGER;                    { first item appended (0 based) }
     pConfig:ConfigPtr;
     value:  INTEGER;

BEGIN
     WITH pSetup^ DO
     BEGIN
          first := count - 1;                  { count is 1 based }
          pConfig := ConfigPtr(theConfig);     { get the config ptr }

          CASE pItem^-first OF
```

```
        myFirstItem:
            BEGIN
              GetDItem(theDialog,first+myFirstItem,itemKind,
             itemHandle,itemRect);
              value := GetCtlValue(ControlHandle(itemHandle))
             value := 1 - value;
             pConfig^.foobar := value;       { stick into config record }
              SetCtlValue(ControlHandle(itemHandle), value); { update
                                                              control }
            END;
         mySecondItem:
            BEGIN
             SysBeep(5);
             FlashMenuBar(0);
            END;
        END; {case}
    END; {with}
END;
```

cmSfilterMsg

Your setup definition code resource should perform function similar to that shown below when it receives `cmSfilterMsg` from the Connection Manager.

When passed to your connection tool, `p1` will contain a pointer to an event record, `p2` will contain a pointer to an item hit in the dialog list, and `p3` will contain a pointer to `magicCookie`.

If the event that was passed to this function was handled, then your connection tool should return 1, otherwise it should return 0.

```
FUNCTION Filter(pSetup: SetupPtr; theCookie: LocalPtr; pEvent:
EventPtr;
     pItem: IntPtr): LONGINT;
BEGIN
    Filter := 0;                                { not hungry }

    IF pEvent^.what = keyDown THEN              { eat all keyDowns }
    BEGIN
         SysBeep(5);
        Filter := 1;                            { processed }
    END;
END;
```

cmScleanupMsg

Your setup definition code resource should perform function similar to that shown below when it receives a `cmcleanupMsg` from the Connection Manager.

When passed to your connection tool, p3 will contain a pointer to the `magicCookie`.

```
PROCEDURE myCleanup(p3: LONGINT);
BEGIN
      DisposPtr( Ptr(p3) );            { dispose of magicCookie }
      p3 := 0;
END;
```

The scripting interface code resource

Your connection tool's scripting interface code resource is responsible for handling the interface between your tool and a scripting language. Your scripting interface code resource will have to handle two messages: `cmMGetMsg` and `cmMSetMsg`.

Your connection tool scripting interface code resource should be a resource of type `cscr` and be able to handle the parameters that are shown below:

```
FUNCTION cscr(hConn: ConnHandle; msg: INTEGER; p1, p2, p3: LONGINT):
LONGINT;
VAR
    pConfig:    ConfigPtr;

BEGIN
    cscr := 0;          { for now }

    CASE msg OF
    cmMgetMsg:
        cscr := LONGINT(GetConfig(hConn));
    cmMsetMsg:
        cscr := SetConfig(hConn, Ptr(p1));
    END; {case}
END;
```

Valid values for `msg` are

```
CONST
      cmMgetMsg          =     0;
      cmMsetMsg          =     1;
```

For each of the messages defined above, `p1`, `p2`, and `p3` take on different meanings, which are discussed under the description of each message.

cmMgetMsg

Your tool will receive `cmMgetMsg` from the Connection Manager when the application requires a string that describes the connection record. The sample code below shows how your application can handle `cmMgetMsg`.

After executing the code necessary to respond to cmMgetMsg, your connection tool should return NIL if there was a problem parsing the configuration string. Otherwise, it should return a pointer to a null-terminated string that contains tokens in American English that represent the configuration record pointer to by config in the connection record.

```
FUNCTION GetConfig(hConn: ConnHandle): Ptr;
VAR
    thePtr:     Ptr;
    pConfig:    ConfigPtr;
    theString,
    string2:    STR255;

BEGIN
    pConfig := ConfigPtr(hConn^^.config);          { get the config record }
    theString := 'FOOBAR ';                        { attribute name is FOOBAR }
    NumToString(pConfig^.foobar, string2);         { get the attribute value }
    theString := CONCAT(string, string2);          { make the config string }
    thePtr := NewPtr(SIZEOF(LENGTH(theString)+1));

    IF thePtr <> NIL THEN
    BEGIN
        BlockMove(Ptr(LONGINT(@theString)+1),
        thePtr, LENGTH(theString));                    { copy it }
        Ptr(LONGINT(thePtr)+LENGTH(theString))^ := 0;    { 0 terminate it }
    END;

    GetConfig := thePtr;                           { bye bye }
END;
```

cmMsetMsg

Your tool will receive cmMsetMsg from the Connection Manager when the application requires your tool to set the fields of the connection record to values that are specified in a string. The Connection Manager will pass a pointer to this string as a parameter to this call. The sample code below shows how your tool can handle cmMsetMsg.

When passed to your connection tool's scripting interface code resource, p1 will be a pointer to an American English null-terminated string that contain tokens representing a configuration record. Your tool should return 0 if there is no problem with the string, or a 1 if processing was aborted.

Your tool should call CMValidate after it has completed executing this routine.

```
FUNCTION SetConfig(hConn: ConnHandle): LONGINT;
VAR
    pConfig     : configptr;
    i, local        : integer;          {loop index}
    myToken     : TokenRecPtr;      {each TokenRec}
    theVal      : longint;
    aTokenPtr   : TokenBlockPtr;    {the whole TokenRec block}
    returnVal   : longint;
```

```
    tokeIndex,                          {string index for TokenRec strings}
    valIndex     : integer;             {string index for value strings}
    tokeStr      : str255;              {TokenRec as string}

begin
     pConfig := ConfigPtr(hConn^^.config);
    returnVal:= -1;                         {always the pessimist}

    {fill it with a whole lotta junk}
     if InitTokenBlock(aTokenPtr) <> noErr then
    begin
        SetConfig:= -1;
        EXIT(SetConfig);                              {abort, abort}
    end;

    aTokenPtr^.source := theStr;                              {what to parse}
     aTokenPtr^.sourceLength := strLen(theStr);   {just how long}

     if IntlTokenize(aTokenPtr) <> tokenOK then        {Thanks, SM 2.0}
    begin
        SetConfig:= -1;
         EXIT(SetConfig);
    end;

    {for every TokenRec}
     for i := 1 to aTokenPtr^.tokenCount do
    begin
         myToken := TokenRecPtr(ord(aTokenPtr^.tokenList) +
                                        (i-1)*sizeOf(TokenRec));
          BlockMove(myToken^.position, Ptr(ORD(@tokeStr)+1), myToken^.length);
         tokeStr[0] := Char(myToken^.length);

        if myToken^.theToken = tokenAlpha then
       begin
            IF tokeStr = 'FOOBAR' THEN
           begin
                {index to next alpha TokenRec that matches values}
                while (i < aTokenPtr^.tokenCount) do
               begin
                   i:= i + 1;
                     myToken := TokenRecPtr(ord(aTokenPtr^.tokenList) +
                                                  (i-1)*sizeOf(TokenRec));
                    BlockMove(myToken^.position,

      Ptr(ORD(@tokeStr)+1), myToken^.length);
                    tokeStr[0] := Char(myToken^.length);

                    if myToken^.theToken = tokenNumeric then
                   begin
                        StringToNum(tokeStr,theVal);
                        pConfig^.foobar := theVal;          {set the new value}
                       returnVal:= 0;                       {no errors, mate}
```

```
                        leave; {while loop}
                    end; {if}
                     if i = aTokenPtr^.tokenCount then
                        returnVal:= 1;
                end; {while}
            end; {interval}
        end; {An Alpha TokenRec}

    end; {for every TokenRec}

     DisposPtr(Ptr(aTokenPtr^.TokenList));
    DisposPtr(Ptr(aTokenPtr));                          {clean it up, boys}
    SetConfig:= returnVal;                              {g'day, mate}

end; {SetConfig}


{    ********************************     }
{    Returns the length of a c-string     }
{    ********************************     }

function  strLen(theString:Ptr):longint;
var
    endPtr   : Ptr;

begin
    endPtr:= theString;
    while endPtr^ <> 0 do        {scan until we find \0 termination}
        endPtr:= ptr(ord(endPtr) + 1);
     strLen:= ord4(endPtr)  - ord4(theString);
end;

{    ******************************************     }
{    initialize the TokenRec block for tokenize call }
{    ******************************************     }

function  InitTokenBlock(var  aTokenPtr:TokenBlockPtr):  longint;
const
    TOKE_MAX     = 10;                {Max # of tokens to support}

var
    itl4         : itl4Handle;

begin
     itl4 := itl4Handle(IUGetIntl(4));
     HLock(Handle(itl4));
     aTokenPtr := TokenBlockPtr(NewPtr(sizeof(TokenBlock)));       {gimme space}
    if aTokenPtr = nil then begin
        InitTokenBlock:= MemError;
        EXIT(InitTokenBlock);
    end;
     aTokenPtr^.tokenList := NewPtr(sizeof(TokenRec) * TOKE_MAX);    {gimme more}
```

```
    if aTokenPtr^.tokenList = nil then
    begin
         InitTokenBlock:= MemError;
         EXIT(InitTokenBlock);
    end;

    aTokenPtr^.tokenLength := TOKE_MAX;
    aTokenPtr^.tokenCount := 0;

    aTokenPtr^.stringList := NIL;                    {unused for my purposes}
    aTokenPtr^.stringLength := 0;
    aTokenPtr^.stringCount := 0;

    aTokenPtr^.doString := false;
    aTokenPtr^.doAppend := false;
    aTokenPtr^.doAlphanumeric := false;
    aTokenPtr^.doNest := false;

    aTokenPtr^.leftDelims[0] := token2Quote;         {a whole lotta junk}
    aTokenPtr^.leftDelims[1] := token2Quote;

    aTokenPtr^.rightDelims[0] := token2Quote;
    aTokenPtr^.rightDelims[1] := token2Quote;

    aTokenPtr^.leftComment[0] := tokenRoot;
    aTokenPtr^.leftComment[1] := tokenRoot;
    aTokenPtr^.leftComment[2] := tokenRoot;
    aTokenPtr^.leftComment[3] := tokenRoot;

    aTokenPtr^.rightComment[0] := tokenRoot;
    aTokenPtr^.rightComment[1] := tokenRoot;
    aTokenPtr^.rightComment[2] := tokenRoot;
    aTokenPtr^.rightComment[3] := tokenRoot;

    aTokenPtr^.escapeCode := tokenEscape;
    aTokenPtr^.decimalCode := tokenPeriod;
    aTokenPtr^.itlResource := Handle(itl4);
    aTokenPtr^.reserved[0] := 0;
    aTokenPtr^.reserved[1] := 0;
    aTokenPtr^.reserved[2] := 0;
    aTokenPtr^.reserved[3] := 0;
    aTokenPtr^.reserved[4] := 0;
    aTokenPtr^.reserved[5] := 0;
    aTokenPtr^.reserved[6] := 0;
    aTokenPtr^.reserved[7] := 0;

    HUnlock(Handle(itl4));
    InitTokenBlock:= noErr;      {pas de problème}
end;  {InitTokenBlock}
```

The localization code resource

Your connection tool's localization code resource is responsible for handling the function necessary to localize your tool. It will have to be able to handle two messages: `CMGetConfig` and `CMSetConfig`.

Your localization code resource should be a resource of type `cloc`; it should be able to handle the parameters that are shown below.

```
FUNCTION    cloc(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

Valid values for `msg` are

```
CONST
      cmL2English          =      0;
      cmL2Intl             =      1;
```

For each of the messages defined above, `p1`, `p2`, and `p3` take on different meanings, which are discussed under the description of each message.

cmL2English and cmL2Intl

Your tool will receive `cmL2English` from the Connection Manager when the application requires your tool to localize a string to English. When the parameters `p1`, `p2`, and `p3` are passed to your tool, `p1` will contain a pointer to a localized null-terminated string that contains tokens representing a configuration record, `p2` will contain a pointer that points to a pointer. Your tool will have to allocate space for this pointer, which contains the American English null-terminated configuration string. `p3` will contain a language identifier, which is defined in the Script Manger documentation.

Your tool will receive `cmL2Intl` from the Connection Manager when the application requires your tool to localize a string to a language other than English. When the parameters `p1`, `p2`, and `p3` are passed to your tool, `p1` will contain a pointer to an American English null-terminated string that contains tokens representing a configuration record, `p2` will contain a pointer to a pointer. Your tool will have to allocate space for this pointer, which contains the localized configuration string. `p3` will contain a language identifier, which is defined in the Script Manger documentation.

The next section of sample code shows how your tool can handle both `cmL2English` and `cmL2Intl`.

After executing the code necessary to respond to a `cmL2English` or `cmL2Intl`, your routine should return `NIL` if there was a Memory Manager error or if the language requested is not available. It should also return any appropriate error code in the status field of the connection record.

```
{    main entry point for cloc resource }
FUNCTION cloc(hConn: ConnHandle; msg: INTEGER; p1, p2, p3: LONGINT): LONGINT;
```

```
TYPE
     PtrPtr = ^Ptr;

VAR
     outPtr: Ptr;
     procID: INTEGER;

begin
     outPtr := PtrPtr(p2)^;           { get output pointer }
     case msg of
          cmL2English:
               cloc := Translate( Ptr(p1),outPtr,p3,verUS);
          cmL2Intl:
               cloc := Translate( Ptr(p1),outPtr,verUS,p3);
     end; {case}
     PtrPtr(p2)^ := outPtr;           { return output pointer }
end;  {mytscrDEF}

{    Translates an input config string from one language to another   }
{    returns 0 if no problem, non zero if there is a problem          }
{    This routine needs to allocate outputStr.                        }
{    if language is not supported, return 0 but leave outputStr NIL   }

function Translate( inputStr: Ptr; var outputStr: Ptr;
       fromLanguage,toLanguage: longint): longint;
BEGIN
end;  {Translate}
```

Summary

Messages

```
CONST
```

```
{       validation code resource messages }
        cmValidateMsg           =       0;
        cmDefaultMsg            =       1;

{       setup code resource messages }
        cmSpreflightMsg         =       0;
        cmSsetupMsg             =       1;
        cmSitemMsg              =       2;
        cmSfilterMsg            =       3;
        cmScleanupMsg           =       4;

{       scripting interface code resource messages }
        cmMgetMsg               =       0;
        cmMsetMsg               =       1;

{       localization interface code resource messages }
        cmL2English             =       0;
        cmL2Intl                =       1;
```

Data structures

```
        SetupPtr                =       ^SetupStruct;
        SetupStruct             =       RECORD
                theDialog       :       DialogPtr;
                count           :       INTEGER;
                theConfig       :       Ptr;
                procID          :       INTEGER
        END;
```

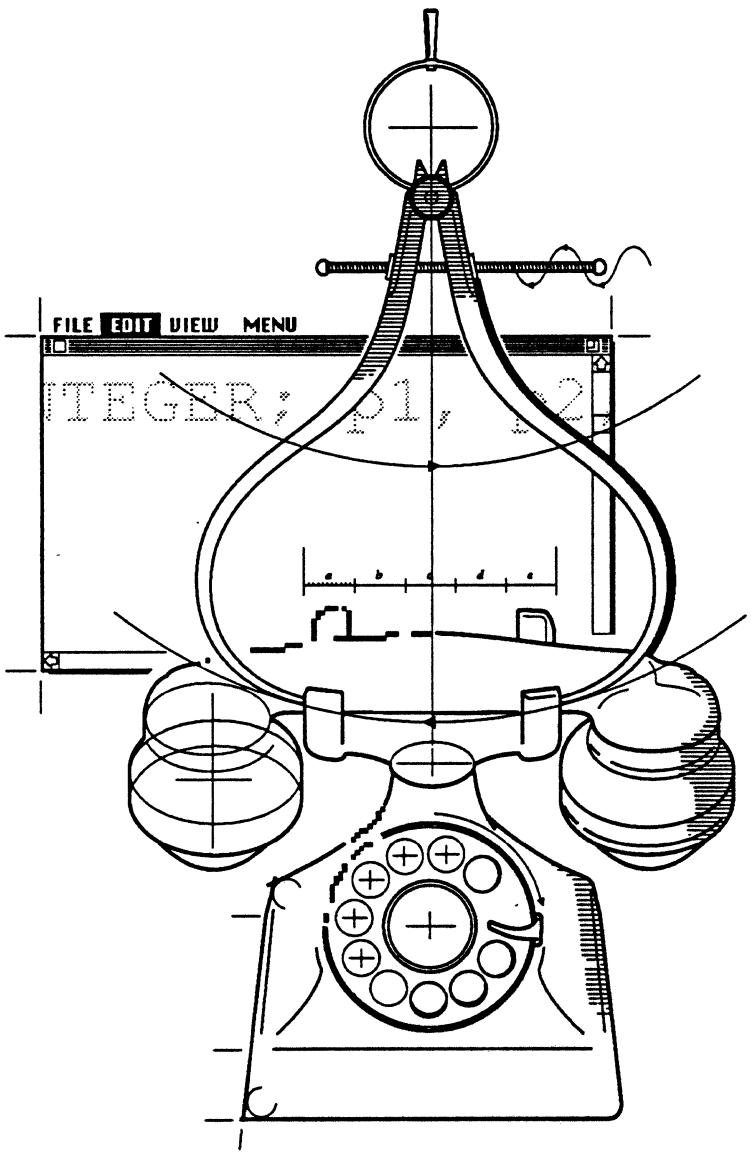
Definition procedures

```
FUNCTION    cdef(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cval(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cset(pSetup: SetupPtr; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cscr(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cloc(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

Resource types

```
type 'cbnd' {
      integer = $$CountOf(TypeArray) - 1;
      array TypeArray {
            literal longint;     /* Type                    */
            integer = $$CountOf(IDArray) - 1;
            wide array IDArray {
                  integer;       /* Local ID                */
                  integer;       /* Actual ID               */
            };
      };
};
```

Chapter 9 **Writing Connection Tools**



About this chapter

This chapter tells you how to write the main code resource for a connection tool. There are at least five other code resources that you will need to include as part of your tool; they are described in a Chapter 8, "Fundamentals of Writing Your Own Tool." You should read that chapter, as well as Chapter 3, "Connection Manager," before reading this chapter.

This chapter describes all the messages, parameters, and data structures that the Connection Manager will pass to your tool's main code resource. Also included in this chapter is sample code (with pseudocode mixed in) that will help you understand what your application should do in response to receiving any of the messages. A summary at the end of the chapter shows you what you should names your six connection tool resources, as well as all the messages the Connection Manager will send to your tool.

Your connection tool's main code resource

The purpose of the main code resource is to parse messages from the Connection Manager and then to branch to a routine that can handle each message. The main code resource should be a resource of type `cdef` and be able to accept the parameters shown below.

```
FUNCTION    cdef(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

The messages accepted by the main code resource, and their associated values, are:

```
CONST
      cmInitMsg           =     0;
      cmDisposeMsg        =     1;
      cmSuspendMsg        =     2;
      cmResumeMsg         =     3;
      cmMenuMsg           =     4;
      cmEventMsg          =     5;
      cmEnvironsMsg       =     6;
      cmActivateMsg       =     50;
      cmDeactivateMsg     =     51;
      cmIdleMsg           =     16;
      cmAbortMsg          =     52;
      cmResetMsg          =     53;
      cmReadMsg           =     100;
      cmWriteMsg          =     101;
      cmStatusMsg         =     102;
      cmListenMsg         =     103;
      cmAcceptMsg         =     104;
      cmCloseMsg          =     105;
      cmOpenMsg           =     106;
      cmBreakMsg          =     107;
      cmIOKillMsg         =     108;
      cmEnvironsMsg       =     1089;
```

For each of the messages defined above, the three parameters `cdef` returns, `p1`, `p2`, and `p3`, take on different meanings. These parameters are described in the following sections that go into detail about how your tool should respond to each incoming message.

cmResetMsg

The Connection Manager will send `cmResetMsg` to your tool when the application requires your tool to reset the connection. The specific state to which your tool should reset the connection is dependent upon the connection protocol.

The sample code below shows you a basic template into which you can code your tool's response to `cmResetMsg`.

```
PROCEDURE myReset(hConn: ConnHandle);
BEGIN
END;
```

cmMenuMsg

The Connection Manager will send `cmMenuMsg` to your tool when a menu event has occurred in the application. When passed to your tool, `p1` will contain the menu ID and `p2` will contain the menu item.

The sample code below shows you a basic template into which you can code your tool's response to cmMenuMsg. When done, your tool should pass back 0 if the menu event was not handled and 1 if it was.

```
FUNCTION myMenu(hConn: ConnHandle; mID: INTEGER; mItem: INTEGER): LONGINT;
BEGIN
      myMenu := 0;                        { pessimism }
      { if mine then
            myMenu := 1;
      }
END;
```

cmListenMsg

An application will call the CMListen routine when it requires your tool to wait for in incoming string of data. When passed to your tool, p1 will contain the address of CompletorRecord and p2 will contain the timeout value in ticks.

The sample code code shows you a basic template into which you can code your tool's response to cmListenMsg. When done, your tool should pass back an appropriate error code.

```
FUNCTION      myListen(hConn: ConnHandle; completor: CompletorRecord;
                       timeout: LONGINT): CMErr;
BEGIN
      myListen := noErr;                        { optimism }

      { if connection is already open, return error condition }
      { if listen is pending, return noErr }

      {     if the value for timeout is 0, try once }
      {     if the value for -1, there is not timeout }
      IF completor.async THEN
      BEGIN
      END
      ELSE
      BEGIN
      END;
END;
```

cmIdleMsg

Your tool will receive cmIdleMsg when the application requires idle time, such as when it needs your tool to check the status of an asynchronous routine. An application can not call CMIdle from interrupt level.

The sample code shows you a template into which you can code your tool's response to cmIdleMsg.

```
PROCEDURE     myIdle(hConn: ConnHandle);
BEGIN
END;
```

cmEventMsg

The Connection Manager will pass `cmEventMsg` to your tool when an event occurred in a window associated with the connection tool. The sample code shows a template into which you can code your tool's response to `cmEventMsg`.

When passed to your tool, `p1` will be a pointer to the event record. The reference constant field of the window record will contain the connection handle.

```
PROCEDURE     myEvent(hConn: ConnHandle; theEvent: EventRecord);
BEGIN
{ process the event }
END;
```

cmAbortMsg

The Connection Manager will pass `cmAbortMsg` to your tool when the application has requested that a pending open or listen be aborted. The sample code below shows a template into which you can code your tool's response to `cmAbortMsg`.

```
PROCEDURE     myAbort(hConn: ConnHandle);
BEGIN
END;
```

cmAcceptMsg

The Connection Manager will pass `cmAcceptMsg` to your tool when the application has called the `cmAccept` routine. When passed to your tool, p1 will contain 1 if your tool should accept the open request or 0 if it should reject it.

The sample code below shows a template into which you can code your tool's response to cmAcceptMsg. When finished, your tool should return an appropriate error code.

```
FUNCTION      myAccept(hConn: ConnHandle; accept: INTEGER): CMErr;
BEGIN
      IF accept = 1 THEN
      BEGIN                                    { accept }
      END
      ELSE
      BEGIN                                    { reject }
      END;
END;
```

cmActivateMsg and cmDeactivateMsg

The Connection Manager will pass `cmActivateMsg` or `cmDeactivateMsg` to your tool when the application requires your tool to perform an action, such as installing or removing a menu from the menu bar.

The sample code below shows a template into which you can code your tool's response to `cmActivateMsg` and `cmDeactivateMsg`. It is possible that your tool will respond identically to each message.

```
PROCEDURE myActivate(hConn: ConnHandle);
BEGIN
END;


PROCEDURE myDeactivate(hConn: ConnHandle);
BEGIN
END;
```

cmSuspendMsg and cmResumeMsg

The Connection Manager will pass `cmSuspendMsg` or `cmResumeMsg` to your tool when the application requires your tool to perform an action, such as installing or removing a menu from the menu bar.

The sample code below shows a template into which you can code your tool's response to `cmSuspendMsg` and `cmResumeMsg` . It is possible that your tool will respond identically to each message.

```
PROCEDURE mySuspend(hConn: ConnHandle);
BEGIN
END;


PROCEDURE myResume(hConn: ConnHandle);
BEGIN
END;
```

cmInitMsg

The connection Manager will pass `cmInitMsg` to your tool after the following sequence of events. When a tool or application calls `CMNew`, the Connection Manager allocates space for the connection record. It then fills in some of the fields based upon information that was passed in the parameters to the call. The Connection Manager fills in the `config` and `oldConfig` fields by calling `CMDefault`. Then the Connection Manager passes `cmInitMsg` to your tool. After your tool has finished responding to a `cmInitMsg`, the Connection Manager calls `CMValidate`.

The sample code below shows how your tool can respond to a `cmInitMsg`. After executing the code necessary to respond to a `cmInitMsg`, your code should pass back an appropriate `OsErr` or `CMErr`.

```
FUNCTION      myInit(hConn: ConnHandle): CMErr;
VAR
      state:       SignedByte;

BEGIN
      myInit := noErr;                                    { optimism }
      state := HGetState(Handle(hConn));              { save handle state }
      HLock(Handle(hConn));                               { lock it down }

      WITH hConn^^ DO
      BEGIN
            flags := BOR(flags, cmData);              { yes we do data }
            IF BAND(flags, cmAttn) <> 0 THEN { turn off attention }
                  flags := BXOR(flags, cmAttn);
            IF BAND(flags, cmCntl) <> 0 THEN { turn off control }
                  flags := BXOR(flags, cmCntl);

            errCode := noErr;                             { optimism reigns }

            { need to check MemErr here }
            bufferArray[CMDataIn] := NewPtr(bufSizes[CMDataIn]);
            bufferArray[CMDataOut] := NewPtr(bufSizes[CMDataOut]);

            private := PrivatePtr(NewPtr(SIZEOF(PrivateData)));
            WITH private^ DO
            BEGIN { fill in private data structure here }
            END;

      END;

      HSetState(Handle(hConn), state);
END;
```

cmDisposeMsg

A tool or application will call `CMDispose` when it requires your tool to disposeof a connection record and all the data structures that are associated with it. If the connection is open when the application calls `CMDispose`, your tool should first make a synchronous call to `CMClose` (with the immediate bit set).

The sample code below shows how your tool can respond to `cmDisposeMsg`. After executing the code necessary to respond to `cmDisposeMsg`, your code should pass back 0 if it was successful or 1 if it was not

```
FUNCTION myDispose(hConn: ConnHandle): CMErr;
VAR
      pPrivate: PrivatePtr;                    { tool privates }

BEGIN
      myDispose := noErr;

{     if the connection is open then call CMClose on it }

      DisposPtr( Ptr(hConn^^.private) );
      DisposPtr( Ptr(hConn^^.bufferArray[CMDataIn]) );
      DisposPtr( Ptr(hConn^^.bufferArray[CMDataOut]) );
END;
```

cmReadMsg and cmWriteMsg

A tool or application will call `CMRead` when it requires your tool to read data from a remote entity. Likewise, a tool or application will call `CMWrite` when it requires your tool to write data to a remote entity. The Connection Manager will handle these calls by passing `cmReadMsg` or `cmWriteMsg` to the appropriate connection tool

If a channel is requested that is not supported by your tool (for example, a read is requested on the attention channel when the attention channel is not supported), your tool should return `cmNotSupported`.

After executing the code necessary to respond to a `cmReadMsg` or `cmWriteMsg`, your tool should pass back both an appropriate `OsErr` or `CMErr`, as well as the following values for `p1`, `p2`, and `p3`: in `p1` a pointer to `dataBuffer`, in `p2` the timeout value in ticks, and in `p3` a pointer to `CompletorRecord`.

When the Connection Manager passes `cmReadMsg` or `cmWriteMsg` to a tool, it will also pass the `dataBuffer` record and the `CompletorRecord`.

Before learning how your tool should respond to `cmReadMsg` or `cmWriteMsg`, first you need to know about the records that are passed along with the message.

The dataBuffer record

`dataBuffer` contains information about where the read or write buffer is located, how many bytes are supposed to be read or written, the channel that is to be used, and an end-of-message flag. Your tool should be able to accomodate the data structure defined here:

```
TYPE
      dataBufferPtr        =     ^dataBuffer;
      dataBuffer           =     RECORD
            thePtr         :     Ptr;          { ptr to buffer }
            count          :     LONGINT;      { # to read/write }
            channel        :     INTEGER;      { channel desired }
            flags          :     BOOLEAN;      { end of message }
      END;
```

These are the valid values for `channel`:

```
CONST
      cmData              =     1;
      cmCntl              =     2;
      cmAttn              =     3;
```

The CompletorRecord record

A `CompletorRecord` record is sent with every message that involves either an asynchronous or synchronous operation. This includes `cmReadMsg` and `cmWriteMsg`. This record allows your tool to execute a completion routine after the read or write has finished.

```
TYPE
      CompletorPtr        =     ^CompletorRecord;
      CompletorRecord     =     RECORD
            async               :     BOOLEAN;
            completionRoutine   :     ProcPtr;
      END;
```

cmReadMsg

```
FUNCTION myRead(hConn: ConnHanlde; dPtr: dataBufferPtr;
                  timeout: LONGINT; completor: CompletorPtr);
BEGIN
{     if connection is not open then return cmNotOpen }

      IF dPtr^.channel <> cmData THEN        { this tool only supports Data }
      { need to check to see if call is requesting proper channel }
      BEGIN
            myRead := cmNotSupported;
            Exit(myRead);
      END;

      {     timeout = -1 is infinite retry }
      {     timeout = 0 then try once }

      IF completor^.async THEN                        { asynchronous read }
      BEGIN
      END
      ELSE
      BEGIN
      END;
END;
```

cmWriteMsg

```
FUNCTION myWrite(hConn: ConnHanlde; dPtr: DataBufferPtr;
                    timeout: LONGINT; completor: CompletorRecord);
BEGIN
{       if connection is not open then return cmNotOpen }

        IF dPtr^.channel <> cmData THEN          { this tool only supports Data }
                { need to check to see if call is requesting proper channel }
        BEGIN
                myWrite := cmNotSupported;
                Exit(myWrite);
        END;

        {       timeout = -1 is infinite retry }
        {       timeout = 0 then try once }

        IF completor.async THEN                          { asynchronous read }
        BEGIN
        END
        ELSE
        BEGIN
        END;
END;
```

cmStatusMsg

The Connection Manager will send `cmStatusMsg` to your tool when an application requires your tool to send it information about a connection.

The sample code below shows how your tool can respond to `cmStatusMsg`. After executing the code necessary to respond to `cmStatusMsg`, your code should pass back both an appropriate `OsErr` or `CMErr`. Also, `p1` should contain a pointer to `Buffersizes`, and `p2` should contain a pointer to a variable that returns the connection status flags.

Connection status flags are a bit field, with each bit corresponding to a particular status attribute. You can find a description of the different status attributes in Chapter 3 "Connection Manager."

```
FUNCTION      myStatus(hConn: ConnHandle; VAR sizes:
                          BufferSizes; VAR flags: LONGINT): CMErr;
BEGIN
      flags := 0;                                    { start at square 1 }
      ### MPW Shell - Missing End command.

      { if connection is open then
      flags := BOR(flags, cmStatusOpen);
      }

      { if read data is available then
            fill in the appropriate field of sizes and

            flags := BOR(flags, CMStatusReadAvail
      }


END;
```

cmOpenMsg

Your tool's main code resource will receive `cmOpenMsg` from the Connection Manager when an application or tool requires your tool to open a connection.

The sample code below shows a template into which you can code your tool's response to `cmOpenMsg`. After executing the code necessary to respond to `cmOpenMsg`, your code should pass back an appropriate `OsErr` or `CMErr`, `p1` should contain a pointer to `CompletorRecord`, and `p2` should contain the timeout value in ticks.

```
FUNCTION      myOpen(hConn: ConnHandle;
                          completor: CompletorPtr; timeout: LONGINT): CMErr;
BEGIN
      myOpen := noErr;                  { optimism }
END;
```

cmCloseMsg

Your tool's main code resource will receive `cmCloseMsg` from the Connection Manager when an application or tool requires your tool to close a connection.

The sample code below shows how your tool can respond to `cmCloseMsg`. When passed to your tool, `p3` will contain the timeout value in ticks. If `p2` is passed in nonzero, abort all outstanding reads or writes. Otherwise, when the outstanding reads and writes complete, close the connection.

After executing the code necessary to respond to a cmCloseMsg, your code should pass back both an appropriate OsErr or CMErr. p1 should contain a pointer to CompletorRecord and p2 should contain 1 if immediate or 0 otherwise.

```
FUNCTION      myClose(hConn: ConnHandle; completor: CompletorPtr; now: LONGINT;
timeout: LONGINT);
BEGIN
{       timeout = 0 then try once
        timeout = -1 the try infinitely
}

        IF now = 1 THEN
        BEGIN
{
        kill pending breaks, reads and or writes
}
        END
        ELSE
        BEGIN
{
        wait for pending breaks, reads and or writes
}
        END;
END;
```

cmBreakMsg

Your tool's main code resource will receive cmBreakMsg when an application or tool requires your tool to effect a break operation upon a connection.

The sample code below shows how your tool can respond to a cmBreakMsg. After executing the code necessary to respond to a cmBreakMsg, your code should pass back an appropriate OsErr or CMErr. p1 should contain duration in ticks and in p2 should contain a pointer to CompletorRecord.

```
FUNCTIOn      myBreak(hConn: ConnHandle; duration: LONGINT;
                                                completor: CompletorPtr):
CMErr;
VAR
      pPrivate: PrivatePtr;
      pConfig: ConfigPtr;
      err: OSErr;
      foo: LONGINT;

BEGIN
      myBreak := noErr;    {optimism}

      pPrivate := PrivatePtr(hConn^^.private);
      pConfig := ConfigPtr(hConn^^.config);

      if ( BAND(pPrivate^.status, cmStatusOpen) == 0 ) THEN        { not open }
      BEGIN
            myBreak :=    cmNotOpen;
            Exit(myBreak);
      END;

      IF (pPrivate^.breakPending) THEN                                    {
break pending }
      BEGIN
            myBreak :=    cmNotOpen;
            Exit(myBreak);
      END;

      IF completor^.async THEN
      BEGIN
            { do it asynchronously }
            { start the break }
            { start a timer (VBL or such) when it completes it will
                  turn off the break and then call the completion routine
                  if necessary }
      END
      ELSE
      BEGIN
            { start the break }
            Delay(duration, foo);
            { end the break }
      END;
END;
```

cmIOKillMsg

Your tool's main code resource will receive `cmIOKillMsg` when a tool or application requires your tool to terminate a pending asynchronous input or output request.

The sample code below shows how your tool can respond to a `cmIOKillMsg`. After executing the necessary code to respond to a `cmIOKillMsg`, your tool should pass back an appropriate `OsErr` or `CMErr`, and `pl` should point to the channel that was affected.

```
FUNCTION      myKill(hConn: ConnHandle; channel: INTEGER): CMErr;
BEGIN
{     make sure that we are using a supported channel }
{     if not supported then
      myKill := cmNotSupported;
}

{     kill pending input or output on the specified channel }

      myKill := noErr;

END;
```

cmEnvironsMsg

The Connection Manager will send `cmEnvironsMsg` to your tool when an application requires your tool to send it information about the connection environment.

The sample code below shows how your tool can respond to `cmEnvironsMsg`.

```
FUNCTION myEnvirons(hConn: ConnHandle; VAR theEnvirons: ConnEnvironRec): CMErr;
VAR
      pConfig:      ConfigPtr;

BEGIN
      pConfig := hConn^^.config;                    { get the config handle }
      myEnvirons := noErr;                          { optimism }

      IF environs.version < 0 THEN
            myEnvirons := envBadVers          { bad environment version }
      ELSE
      BEGIN
            IF environs.version > 1 THEN            { too advanaced for me }
                  myEnvirons := envVersTooBig;      { but give it a whirl }

            WITH environs DO
            BEGIN
                  CASE pConfig^.dataBits OF
                  data5:
                        dataBits := 5;
                  data6:
                        dataBits := 6;
                  data7:
                        dataBits := 7;
                  data8:
                        dataBits := 8;
                  END; {case}

                  baudrate := pConfig^.baudrate;
                  swFlowControl := ((pConfig^.shaker.fInX) AND
                  (pConfig^.shaker.fXOn));
                  hwFlowControl := ((pConfig^.shaker.fDTR) OR
                  (pConfig^.shaker.fCTS));
                  flags := 0;          { no special flags supported }
                  channels := cmData; { data channel only }
            END;
      END;
END;
```

Summary

Constants

```
CONST
```

```
{       main definition procedure messages }
        cmInitMsg           =       0;
        cmDisposeMsg        =       1;
        cmSuspendMsg        =       2;
        cmResumeMsg         =       3;
        cmMenuMsg           =       4;
        cmEventMsg          =       5;
        cmEnvironsMsg       =       6;
        cmActivateMsg       =       50;
        cmDeactivateMsg     =       51;
        cmIdleMsg           =       16;
        cmAbortMsg          =       52;
        cmResetMsg          =       53;
        cmReadMsg           =       100;
        cmWriteMsg          =       101;
        cmStatusMsg         =       102;
        cmListenMsg         =       103;
        cmAcceptMsg         =       104;
        cmCloseMsg          =       105;
        cmOpenMsg           =       106;
        cmBreakMsg          =       107;
        cmIOKillMsg         =       108;
        cmEnvironsMsg       =       109;

{       validation code resource messages }
        cmValidateMsg       =       0;
        cmDefaultMsg        =       1;

{       setup code resource messages }
        cmSpreflightMsg     =       0;
        cmSsetupMsg         =       1;
        cmSitemMsg          =       2;
        cmSfilterMsg        =       3;
        cmScleanupMsg       =       4;

{       scripting interface code resource messages }
        cmMgetMsg           =       0;
        cmMsetMsg           =       1;

{       localization interface code resource messages }
        cmL2English         =       0;
        cmL2Intl            =       1;
```

Data structures

```
TYPE
      dataBufferPtr        =      ^dataBuffer;
      dataBuffer           =      RECORD
            thePtr         :      Ptr;
            count          :      LONGINT;
            channel        :      INTEGER;
            flags          :      BOOLEAN;
      END;

      CompletorPtr         =      ^CompletorRecord;
      CompletorRecord      =      RECORD
            async          :      BOOLEAN;
            completionRoutine
                           :      ProcPtr;
      END;
```

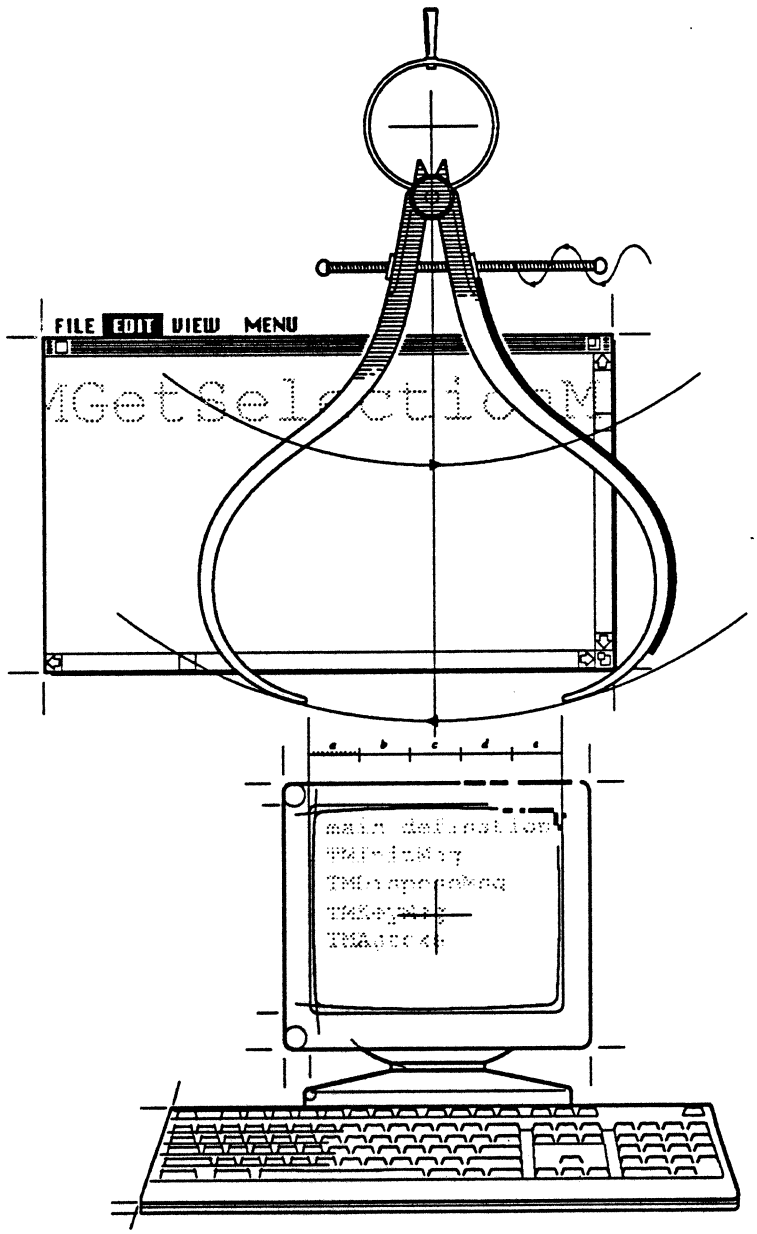
Definition procedures

```
FUNCTION    cdef(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cval(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cset(pSetup: SetupPtr; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cscr(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    cloc(hConn: ConnHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

Resource types

```
type 'cbnd' {
      integer = $$CountOf(TypeArray) - 1;
      array TypeArray {
            literal longint;     /* Type                    */
            integer = $$CountOf(IDArray) - 1;
            wide array IDArray {
                  integer;       /* Local ID                */
                  integer;       /* Actual ID               */
            };
      };
};
```

Chapter 10 **Writing Terminal Tools**



About this chapter

This chapter tells you how to write the main code resource for a terminal tool. You will need to include six code resources you as part of your tool; they are described in Chapter 8, "Fundamentals of Writing Your Own Tool." You should read that chapter, as well as Chapter 5, "Terminal Manager," before reading this chapter.

This chapter describes all the messages, parameters, and data structures that the Terminal Manager will pass to your tool's main code resource. Also included in this chapter is sample code (with pseudocode mixed in) that will help you understand what your application should do in response to receiving any of the messages. A summary at the end of the chapter shows you both how to name your terminal tool resources as well as all of the messages that the Terminal Manager will send to your tool.

Your terminal tool's main code resource

The purpose of the main code resource is to parse messages from the Terminal Manager and then to branch to a routine that can handle each message. The main code resource should be a resource of type `tdef` and be able to accept the parameters shown below.

```
FUNCTION    tdef(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

The accepted messages are:

```
CONST
    tmInitMsg           =    0;
    tmDisposeMsg        =    1;
    tmSuspendMsg        =    2;
    tmResume            =    3;
    tmMenuMsg           =    4;
    tmEventMsg          =    5;
    tmActivateMsg       =    50;
    tmDeactivateMsg     =    51;
    tmIdleMsg           =    52;
    tmResetMsg          =    54;
    tmKeyMsg            =    100;
    tmAutokeyMsg        =    101;
    tmStreamMsg         =    102;
    tmResizeMsg         =    103;
    tmUpdateMsg         =    104;
    tmClickMsg          =    105;
    tmGetSelectionMsg   =    106;
    tmSetSelectionMsg   =    107;
    tmScrollMsg         =    108;
```

```
tmClearMsg          =       109;
tmGetLineMsg        =       110;
tmPaintMsg          =       111;
tmCursorMsg         =       112;
tmGetEnvironsMsg    =       113;
tmDoTermKeyMsg      =       114;
tmCountTermKeyMsg   =       115;
tmGetINDTermKeyMsg  =       116;
```

tmInitMsg

The Terminal Manager will pass `tmInitMsg` to your tool after the following sequence of events. When a tool or application calls `TMNew`, the Terminal Manager allocates space for the terminal record. It then fills in some of the fields based upon information that was passed in the parameters to the call. The Terminal Manager fills in the `config` and `oldConfig` fields by calling `TMDefault`. Then the Terminal Manager passes `tmInitMsg` to your tool. After your tool has finished responding to `tmInitMsg`, the Terminal Manager calls `TMValidate`.

The sample code below shows how your tool can respond to a `tmInitMsg`. After executing the code necessary to respond to a `tmInitMsg`, your code should pass back an appropriate `OsErr` or `TMErr`.

```
FUNCTION myInit(hTerm: TermHandle): LONGINT;
VAR
      pPrivate:    PrivatePtr;

BEGIN
      myInit := 0;                              { optimism }

      pPrivate := PrivatePtr(NewPtr(SIZEOF(PrivateRecord)));
      hTerm^^.private := pPrivate;
      IF pPrivate = NIL THEN
            myInit := MemError;
END;
```

tmDisposeMsg

A tool or application will call `TMDispose` when it requires your tool to get rid of a terminal record and all the data structures that are associated with it.

The sample code below shows a template into which you can code your tool's response to `tmDisposeMsg`. After executing the code necessary to respond to `tmDisposeMsg`, your code should pass back 0 if it was successful or 1 if it was not.

```
PROCEDURE myDispose(hTerm: TermHandle);
BEGIN
      DisposPtr( Ptr(hTerm^^.private) );
```

```
END;
```

tmKeyMsg

Your tool will receive `tmKeyMsg` in response to a key down, key up, or autokey event in the application. Your tool should be able to handle these events, as well as pasting text from the keyboard. The sample code below shows how your tool can respond to these messages.

When passed to your tool, `p1` will point to the event record associated with the event. In the case of pasting text, the `keyCode` field of the event record contains 0; only `charCode` contains information.

```
PROCEDURE myKey(hTerm: TermHandle; pEvent: EventPtr);
VAR
      theChar: Char;
      ticket: INTEGER;
      foo: LONGINT;

BEGIN
      theChar := Char(BAND(pEvent^.message, charCodeMask));

      ticket := BitShift(theChar, -8);              { shift 8 bits to the left }

      foo := CallSendProc(Ptr(@ticket), 1, hTerm^^.refCon, 0, hTerm^^.sendProc);
                              { this returns # characters sent
                              passed in pointer to buffer
                              passed in length of data to send
                              passed in terminal record refcon
                              passed in end of message flag (0)
                                                                  }
END;
```

tmStreamMsg

The Terminal Manager will pass `tmStreamMsg` to your tool when the application has requested the `TMStream` routine. When passed to your tool, `p1` will point to the buffer of incoming data and `p2` will contain the length of the buffer in bytes. The sample code below shows a template into which you can code your tool's response to `tmStreamMsg`.

After executing the code necessary to respond to a `tmStreamMsg`, your tool should return the number of characters it processed.

```
FUNCTION myStream(hTerm: TermHandle; theBuffer: Ptr; theLength: LONGINT): LONGINT;
BEGIN
      myStream := theLength;                        { optimism }
```

```
        { process character interpreting into screen buffer and drawing if necessary }
END;
```

tmActivateMsg and tmResumeMsg

Your tool will receive `tmActiveateMsg` or `tmResumeMsg` when the application requires your tool to process an activate event (such as inserting menus into the menu bar, modifying a selection, or making the cursor blink) for a window that belongs to the Terminal Manager. The sample code below shows a template into which you can code your tool's response to `tmActivateMsg` or `tmResumeMsg`.

```
PROCEDURE myActivate(hTerm: TermHandle);
BEGIN
END;

PROCEDURE myResume(hTerm: TermHandle);
BEGIN
END;
```

tmDeactivateMsg and tmSuspendMsg

Your tool will receive `tmDeactiveateMsg` or `tmSuspendMsg` when the application requires your tool to process a deactivate event (such as removing a menu from the menu bar, modifying a selection, or making a cursor stop blinking) for a window that belongs to the Terminal Manager. The sample code below shows how your tool can respond to `tmDeactivateMsg` or `tmSuspendMsg`.

```
PROCEDURE myDeactivate(hTerm: TermHandle);
BEGIN
END;

PROCEDURE mySuspend(hTerm: TermHandle);
BEGIN
END;
```

tmResizeMsg

Your tool will receive from the Terminal Manager `tmResizeMsg` when the application requires your tool to resize the terminal emulation window. When passed to your tool, `p1` will point to the rectangle that describes the new view. The code sample below shows how your application can handle `tmResizeMsg`.

```
PROCEDURE myResize(hTerm: TermHandle; p1: RectPtr);
BEGIN
```

```
        { should recalculate visible rows and columns, etc...}
END;
```

tmIdleMsg

Your tool will receive `tmIdleMsg` from the Terminal Manager when the application requires your tool to make the cursor blink or when your tool should process a `tmClickMsg`. The sample code below shows a template into which you can code your tool's response to `tmIdleMsg`.

```
PROCEDURE myReset(hTerm: TermHandle);
BEGIN
END;
```

tmUpdateMsg

Your tool will receive `tmUpdateMsg` from the Terminal Manager when the application requires your tool to update the terminal emulation window. When passed to your tool, `p1` will be a handle to the region that needs to be updated. The sample code below shows a template into which you can code your tool's response to `tmUpdateMsg`.

```
PROCEDURE myUpdate(hTerm: TermHandle; updateRgn: RgnHandle);
BEGIN
END;
```

tmClickMsg

Your tool will receive `tmClickMsg` from the Terminal Manager when the application requires your tool to handle a mouse down event in the terminal emulation window. Your tool should support placing and dragging the cursor. When passed to your tool, `p1` will contain a pointer to the event record.

The sample code below shows a template into which you can code your tool's response to `tmClickMsg`.

```
PROCEDURE myClick(hTerm: TermHandle; p1: EventPtr);
BEGIN
END;
```

tmMenuMsg

Your tool will receive `tmMenuMsg` from the Terminal Manager when the application has selected an item from a menu that belongs to your terminal tool. When passed to your tool, `p1` will contain the menu ID and `p2` will contain the menu item. The sample code below shows a template into which you can code your tool's response to `tmMenuMsg`.

After your tool has performed the function necessary to handle a `tmmenuMsg`, it should return 0 if it did not handle the menu event and 1 if it did.

```
FUNCTION      myMenu(hTerm: TermHandle; mID: INTEGER; mItem: INTEGER): LONGINT;
BEGIN
      myMenu := 0;                        { pessimism - not processed }

      { if mine then myMenu := 1 }

END;
```

tmGetSelectionMsg

Your tool needs to be able to handle `tmGetSelectionMsg` to support cut and copy operations in the terminal emulation window. The sample code below shows a template into which you can code your tool's response to do this..

After performing the necessary function to respond to `tmGetSelectionMsg`, your tool should pass back a handle to the data in `p1`, a pointer to the size of the data in `p2`, and a pointer to the scrap type (`ResType`) in `p3`. Your tool should also return an error code, if appropriate.

```
FUNCTION      myGetSelect(hTerm: TermHandle; theHandle: Handle;
                          theType: ResType): LONGINT;
BEGIN
      myGetSelect := noErr;                                    { optimism }
END;
```

tmSetSelectionMsg

An application will call `tmSetSelection` when it requires your tool to highlight an area of the terminal emulation window. When passsed to your tool, `p1` will point to field that needs to be highlighed and `p2` will describe the type of selection. The code below shows a template into which you can code your tool's response to `tmSetSelectionMsg`.

```
PROCEDURE     mySetSelect(hTerm: TermHandle; selRect: Rect; selType: INTEGER);
BEGIN
END;
```

tmScrollMsg

An application will call `tmScroll` when it requires your tool to scroll the screen either horizontally or vertically. When passed to your tool, `p1` will contain the amount of horizontal

scrolling and p2 will contain the amount of vertical scrolling. The code below shows a template into which you can code your tool's response to tmScrollMsg.

```
PROCEDURE myScroll(hTerm: TermHandle; dH, dV; INTEGER);
BEGIN
END;
```

tmResetMsg

Your tool will receive tmResetMsg when the application requires your tool to reset the terminal emulation window. This reset operation should purge all local screen buffers and be a local operation.

The code sample below shows a template into which you can code your tool's response to tmResetMsg.

```
PROCEDURE myReset(hTerm: TermHandle);
BEGIN
END;
```

tmClearMsg

Your tool will receive tmClearMsg when the application needs your tool to clear the terminal emulation window. This clear operation should purge all local screen buffers and be a local operation.

The code sample below shows a template into which you can code your tool's response to tmClearMsg.

```
PROCEDURE myClear(hTerm: TermHandle);
BEGIN
END;
```

tmGetLineMsg

An application will call TMGetline when it requires your tool to send it a TermDataBlock (which contains the data, character attributes, and line attributes) for a specified line. When passed to your tool, p1 will contain the line number.

The sample code below shows a template into which you can code your tool's response to tmGetLineMsg. Your tool should return in p2 a pointer to the TermDataBlock for the requested line.

```
PROCEDURE    myGetLine(hTerm: TermHandle; lineNo: INTEGER; VAR theTermData:TermDataBlock);
BEGIN
```

```
END;
```

tmPaintMsg

An application will call `TMPaint` when it requires your tool to display the contents of a `TermDataBlock`. When passed to your tool, `p1` will point to the `TermDataBlock` and `p2` will point to the rectangle into which your tool is to display the line.

If `theTermData.theData` is a handle to plain text (not styled), your tool can calculate the number of characters to paint by calling `GetHandleSize`. If your tool requires the data in `theTermData` after it passes control back to the calling application, it must make a copy of this data, since the application may change or destroy it.

The sample code below shows a template into which you can code your tool's response to `tmPaintMsg`.

```
PROCEDURE     myPaintLine(hTerm: TermHandle; theTermData:TermDataBlock; theRect: Rect);
BEGIN
END;
```

tmCursorMsg

An application will call `TMCursor` when it requires your tool to pass it the current location of the cursor. When passed to your tool, `p1` will specify the type of cursor.

The sample code below shows a template into which you can code your tool's response to `tmCursorMsg`. Your tool should return the current cursor position.

```
FUNCTION myGetCursor(hTerm: TermHandle; cursType: LONGINT): Point;
BEGIN

END;
```

tmGetEnvironsMsg

Your tool will receive `tmGetEnvironsMsg` when the application has called the `TMGetTermEnvirons` routine. When passed to your tool, `p1` will point to the `TermEnvironRec`; your tool should fill in this record.

The sample code below shows a template into which you can code your tool's response to `tmGetEnvironsMsg`.

```
FUNCTION myEnvirons(hTerm: TermHandle; VAR theEnvirons: TermEnvironRec): TMErr;
BEGIN
      IF theEnvirons.version < 0 THEN   { check version number of record }
```

```
        BEGIN
                myEnvirons := envBadVers;
                Exit(myEnvirons);
        END;

        IF theEnvirons.version > 0 THEN
                myEnvirons := envVersTooBig
        ELSE
                myEnvirons := noErr;

        WITH myEnvirons DO                          { fill her up… }
        BEGIN
        END;

END;
```

tmEventMsg

The Terminal Manager will pass `tmEventMsg` to your tool when an event occurred in a window associated with the terminal tool. The sample code shows a template into which you can code your tool's response to `tmEventMsg`.

When passed to your tool, `p1` will be a pointer to the event record. The reference constant field of the window record will contain the connection handle.

```
PROCEDURE      myEvent(hTerm: TermHandle; theEvent: EventRecord);
BEGIN
{ process the event }
END;
```

tmDoTermKeyMsg

Your tool will receive `tmDoTermKeyMsg` when the application has called the `TMDoTermKey` routine. When passed to your tool, `p1` will point to a string that corresponds to the key that was pressed. For example, if the user pressed the PF1 key, the string will containg "PF1." If there is no key that corresponds to the string , your tool should do nothing.

The code sample below shows a template into which you can code your tool's response to `tmDoTermKeyMsg`. When completed, your tool should pass back 0 if it understood the string or 1 if it did not.

```
PROCEDURE myDoTermKey(hTerm: TermHandle; p1: StringPtr);
BEGIN
        IF p1^ = 'F13' THEN
                DoF13(hTerm);
        IF p1^ = 'F14' THEN
                DoF14(hTerm);
END;
```

tmCountTermKeyMsg

Your tool will receive `tmCountTermKeyMsg` when the application requires your tool to pass it the number of special terminal key names that it supports.

The sample code below shows how your tool can respond to `tmCountTermKeyMsg`.

```
FUNCTION CountTermKey(hTerm: hTerm): LONGINT;
BEGIN
      CountTermKey := 2;
END;
```

tmGetINDTermKeyMsg

The Terminal Manager will pass to your tool `tmGetINDTermKeyMsg` when the application requires your tool to pass it the name of a special terminal key (for example, PF1, PA1, or DUP). When passed to your tool, `p1` contains the index (number) of the key.

The code sample below shows a template into which you can code your tool's response to `tmGetINDTermKeyMsg`. When completed, your tool should pass back a pointer to a STR255 return value that describes the key, or a pointer to an empty string if the index is invalid.

```
PROCEDURE myGetIndTermKey(hTerm: TermHandle; p1: LONGINT; p2: StringPtr);
BEGIN
      p2^ := '';                          { pessimism }

      CASE p1 OF
      1:
            p2^ := 'F13';
      2:
            p2^ := 'F14';
      END;
END;
```

Summary

Constants

```
CONST
{     main definition procedure messages }
      tmInitMsg            =     0;
      tmDisposeMsg         =     1;
      tmKeyMsg             =     3;
      tmAutokeyMsg         =     4;
      tmStreamMsg          =     5;
```

```
        tmActivateMsg       =       6;
        tmDeactivateMsg     =       7;
        tmSuspendMsg        =       8;
        tmResumeMsg         =       9;
        tmResizeMsg         =       10;
        tmIdleMsg           =       11;
        tmUpdateMsg         =       12;
        tmClickMsg          =       13;
        tmMenuMsg           =       14;
        tmGetSelectionMsg   =       15;
        tmSetSelectionMsg   =       16;
        tmScrollMsg         =       17;
        tmResetMsg          =       18;
        tmClearMsg          =       19;
        tmGetLineMsg        =       20;
        tmPaintMsg          =       21;
        tmCursorMsg         =       22;
        tmGetEnvironsMsg    =       23;
        tmEventMsg          =       24;
        tmDoTermKeyMsg      =       25;
        tmCountTermKeyMsg   =       26;
        tmGetINDTermKeyMsg  =       27;


{       validation code resource messages }
        tmValidateMsg       =       0;
        tmDefaultMsg        =       1;

{       setup code resource messages }
        tmSpreflightMsg     =       0;
        tmSsetupMsg         =       1;
        tmSitemMsg          =       2;
        tmSfilterMsg        =       3;
        tmScleanupMsg       =       4;

{       scripting interface code resource messages }
        tmMgetMsg           =       0;
        tmMsetMsg           =       1;

{       localization interface code resource messages }
        tmL2English         =       0;
        tmL2Intl            =       1;
```

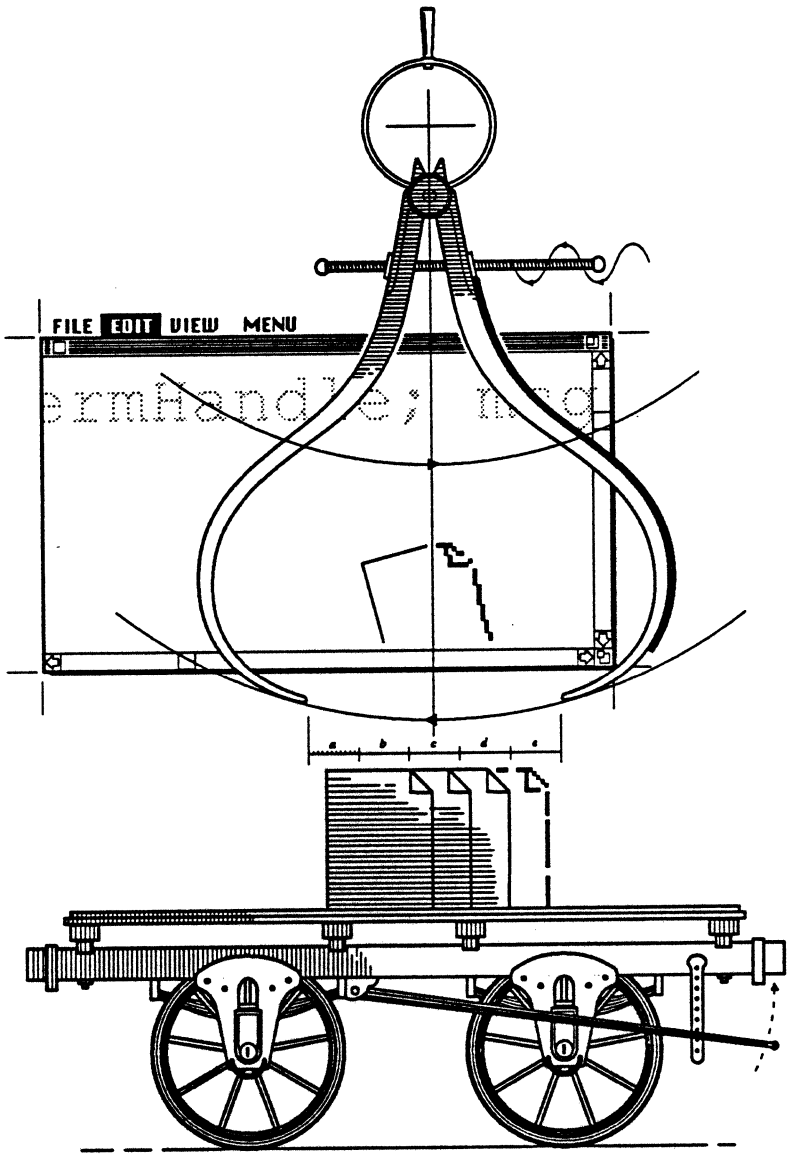
Definition procedures

```
FUNCTION    tdef(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    tval(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    tset(pSetup: SetupPtr; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    tscr(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    tloc(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

Resource types.i.Terminal Tools:Resource Types;

```
type 'tbnd' {
      integer = $$CountOf(TypeArray) - 1;
      array TypeArray {
            literal longint;          /* Type                    */
            integer = $$CountOf(IDArray) - 1;
            wide array IDArray {
                  integer;            /* Local ID                */
                  integer;            /* Actual ID               */
            };
      };
};

type 'tver' as 'vers';
```

Chapter 11 **Writing File Transfer Tools**



About this chapter

This chapter tells you how to write the main code resource for a file transfer tool. You will need to include six other code resources as part of your tool; they are described in Chapter 8, "Fundamentals of Writing Your Own Tool," as well as Chapter 6, "File Transfer Manager," before reading this chapter.

This chapter describes all the messages, parameters, and data structures that the File Transfer Manager will pass to your tool's main code resource. Also included in this chapter is sample code (with pseudocode mixed in) that will help you understand what your application should do in response to receiving any of the messages. A summary at the end of this chapter the shows you the names all six of your file transfer tool resources, as well as all of the messages that the File Transfer Manager will send to your tool.

Your file transfer tool's main code resource

The purpose of the main code resource is to parse messages from the File Transfer Manager and then to branch to a routine that can handle each message. The main code resource should be a resource of type `fdef` and be able to accept the parameters shown below.

```
FUNCTION    fdef(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

The accepted messages are:

```
CONST
      ftInitMsg          =     0;
      ftDisposeMsg       =     1;
      ftSuspendMsg       =     2;
      ftResumeMsg        =     3;
      ftMenuMsg          =     4;
      ftEventMsg         =     5;
      ftActivate         =     50;
      ftDeactivate       =     51;
      ftAbort            =     52;
      ftStartMsg         =     100;
      ftCleanupMsg       =     101;
      ftExecMsg          =     102;
```

For each of the messages defined above, the three parameters that `fdef` returns, `p1`, `p2`, and `p3`, take on different meanings. These parameters are described in the following sections, which go into detail about how your tool should respond to each incoming message.

ftInitMsg

The File Transfer Manager will pass `ftInitMsg` to your tool after the following sequence of events. When a tool or application calls `FTNew`, the File Transfer Manager allocates space for the file transfer record. It then fills in some of the fields based upon information that was passed in the parameters to the call. The File Transfer Manager fills in the `config` and `oldConfig` fields by calling `FTDefault`. Then the File Transfer Manager passes `ftIniftMsg` to your tool. After your tool has finished responding to a `ftIniftMsg`, the File transfer Manager calls `FTValidate`.

After executing the code necessary to respond to a `ftInitmsg`, a sample of which is shown below, your code should pass back an appropriate `OsErr` or `FTErr`.

```
FUNCTION      myInit(hFT: FTHandle): CMErr;
VAR
      state:        SignedByte;

BEGIN
      myInit := noErr;                                    { optimism }
      state := HGetState(Handle(hFT));          { save handle state }
      HLock(Handle(hFT));                                 { lock it down }

      WITH hFT^^ DO
      BEGIN
            errCode := noErr;                             { optimism reigns }

            private := PrivatePtr(NewPtr(SIZEOF(PrivateData)));
            WITH private^ DO
            BEGIN { fill in private data structure here }
            END;
      END;

      HSetState(Handle(hFT), state);
END;
```

ftDisposeMsg

A tool or application will call `ftDispose` when it requires your tool to dispose of a file transfer record and all the data structures that are associated with it.

The sample code below shows a template into which you can code your tool's response to `ftDisposeMsg`. After executing the code necessary to respond to `ftDisposeMsg`, your code should pass back 0 if it was successful or 1 if it was not.

```
PROCEDURE myDispose(hFT: FTHandle);
VAR
      err: FTErr;

BEGIN
      IF BAND(hFT^^.flags, FTIsFTMode) THEN
      BEGIN
            err := FTAbort(hFT);                              { abort FT in
progress }
            FTExec(hFT);                                      { for cleanup }
      END;

      DisposPtr( Ptr(hFT^^.private) );
END;
```

ftStartMsg

Your tool will receive `ftStartMsg` from the File Transfer Manager when the application requires your tool to start a file transfer. The sample code below shows a template into which you can code your tool's response to `ftStartMsg`.

After executing the code necessary to respond to an `ftStartMsg`, your tool should pass back 0 if it was successful and 1 if it was not.

```
FUNCTION FTStartup(hFT: FTHandle): FTErr;
BEGIN
      FTStart := noErr;                                                {
optimism }

      WITH hFT^^ DO
      BEGIN
            errCode := 0;
            flags := BOR(flags, ftIsFTMode);                  { file transfer in
progress }

      END;

END;
```

ftCleanupMsg

The File Transfer Manager will send `ftCleanupMsg` to your tool when the application has called the `ftSetupCleanup` routine. Your tool should respond to this message by confirming the file name of the received file, closing appropriate files, resetting flags, an releasing any temporary memory that it had allocated.

The sample code below shows a template into which you can code your tool's response to `ftCleanupMsg`.

```
PROCEDURE myCleanup(hFT: FTHandle);
BEGIN
      WITH hFT^^ DO
    . BEGIN
            flags := BAND(flags, BNOT(ftIsFTMode) );                    { turn
off FTMode bit }
      END;
END;
```

ftExecMsg

An application will call `ftExec` to provide time for buffers to be either filled or emptied during file transfer, depending upon if a file is being sent or received. The sample code below shows a template into which you can code your tool's response to `ftExecMsg`.

```
PROCEDURE FTExec(hFT: FTHandle);
BEGIN
      { called when file transfer is in progress so do your
            stuff here...
      }
END;
```

ftAbortMsg

Your tool will receive `ftAbortMsg` from the File Transfer Manager when the application requires your tool to abort a file transfer. The sample code below a template into which you can code your tool's response to `ftAbortMsg`.

If your tool is unable to successfully abort, it should pass back an appropriate error code.

```
FUNCTION FTAbort(hFT: FTHandle): FTErr;
BEGIN
      WITH hFT^^ DO
            flags := BAND(flags, BNOT(ftIsFTMode) );        { turn off bit }

      { abort the file transfer in progress here }
END;
```

ftActivateMsg and ftResumeMsg

Your tool will receive `ftActiveateMsg` or `ftResumeMsg` when the application requires your tool to process an activate event (such as inserting menus into the menu bar, modifying a selection, or making the cursor blink) for a window that belongs to the File Transfer Manager. The sample code below shows a template into which you can code your tool's response to `ftActivateMsg` or `ftResumeMsg`.

```
PROCEDURE myActivate(hFT: FTHandle);
BEGIN
```

```
END;
{
       p1, p2, p3 are ignored

       This routine may perform actions such as removing a menu into the
       menubar.
}


PROCEDURE myResume(hFT: FTHandle);
BEGIN

END;

{
       p1, p2, p3 are ignored

       This routine may perform actions such as removing a menu into the
       menubar.  This routine may perform the same actions as myDeactivate
}
```

ftDeactivateMsg and ftSuspendMsg

Your tool will receive `ftDeactiveateMsg` or `ftSuspendMsg` when the applicaiton requires your tool to process a deactivate event (such as removing a menu from the menu bar, modifying a selection, or making a cursor stop blinking) for a window that belongs to the File Transfer Manager. The sample code below shows how your tool can respond to a `ftDeactivateMsg` message or `ftSuspendMsg`.

```
PROCEDURE myDeactivate(hFT: FTHandle);
BEGIN

END;

PROCEDURE mySuspend(hFT: FTHandle);
BEGIN

END;
```

ftMenuMsg

The File Transfer Manager will send CM`MenuMsg` to your tool when a menu event has occurred in the application. When passed to your tool, `p1` will contain the menu ID and `p2` will contain the menu item.

The sample code below shows you a basic template into which you can code your tool's response to `CMMenuMsg`. When done, your tool should pass back 0 if the menu event was not handled and 1 if it was.

```
FUNCTION myMenu(hFT: FTHandle; mID: INTEGER; mItem: INTEGER): LONGINT;
BEGIN
      myMenu := 0;                        { pessimism }
      { if mine then
            myMenu := 1;
      }
END;
```

ftEventMsg

Your tool will receive `ftEventMsg` from the File Transfer Manager when an event has occurred in the application. When passed to your tool, `p1` will point the the event record, in which reference constant field contains the file transfer handle.

The sample code below shows a template into which you can code your tool's response to `ftEventMsg`.

```
PROCEDURE     myEvent(hFT: FTHandle; theEvent: EventRecord);
BEGIN
      { process the event }
END;
```

Summary

Constants

```
CONST
{       messages for main definition procedure }
        ftInitMsg               =       0;
        ftDisposeMsg            =       1;
        ftSuspendMsg            =       2;
        ftResumeMsg             =       3;
        ftMenuMsg               =       4;
        ftEventMsg              =       5;
        ftActivate              =       50;
        ftDeactivate            =       51;
        ftAbort                 =       52;
        ftStartMsg              =       100;
        ftCleanupMsg            =       101;
        ftExecMsg               =       102;


{       validation code resource messages }
        ftValidateMsg           =       0;
        ftDefaultMsg            =       1;

{       setup code resource messages }
        ftXpreflightMsg         =       0;
        ftXsetupMsg             =       1;
        ftXitemMsg              =       2;
        ftXfilterMsg            =       3;
        ftXcleanupMsg           =       4;

{       scripting interface code resource messages }
        ftMgetMsg               =       0;
        ftMsetMsg               =       1;

{       localization interface code resource messages }
        ftL2English             =       0;
        ftL2Intl                =       1;
```

Data types

```
TYPE
      SetupPtr             =      ^SetupStruct;
      SetupStruct          =      RECORD
            theDialog      :      DialogPtr;
            count          :      INTEGER;
            theConfig      :      Ptr;
      END;
```

Definition procedures

```
FUNCTION    fdef(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    fval(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    fset(pSetup: SetupPtr; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    fscr(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;

FUNCTION    floc(hTerm: TermHandle; msg: INTEGER; p1, p2, p3:
            LONGINT) : LONGINT;
```

Resource types.i.File Transfer Tools:Resource Types;

```
type 'fbnd' {
      integer = $$CountOf(TypeArray) - 1;
      array TypeArray {
            literal longint;           /* Type                    */
            integer = $$CountOf(IDArray) - 1;
            wide array IDArray {
                  integer;             /* Local ID                */
                  integer;             /* Actual ID               */
            };
      };
};
```

Appendix A **Guidelines for Communications Tools**

About this appendix

This appendix contains software design and human interface guidelines for communications tools. Guidelines are necessary so that tools from different communications environments can work with each other to extend the users' reach from the desktop to the world of networking and communications. The guidelines presented in this appendix, while not hard and fast rules, will help insure that your tool can interface with future releases of Special K, with other tools, and also with applications that use Special K.

This appendix starts out with the design goals your tool should implement. Then it discusses human interface considerations. At the end of this appendix are hardware and software compatibility requirements.

To fully understand what is being discussed in this chapter, you should first read Chapter 8, "Fundamentals of Writing a Tool," and at least one of the following chapters: Chapter 9, "Writing a Connection Tool," Chapter 10, "Writing a Terminal Tool," or Chapter 11, "Writing a File Transfer Tool."

Design goals

You should design your tool such that it is:

- Self-contained: Your tool should contain all the resources it needs in its bundle resource, and not need to make use of other tools or applications.
- Task-specific: Your tool should be either a connection, terminal, or file transfer tool. It should respond to all the messages that the manager sends to it, but not to any messages that Special K intends a different tool to respond to. For instance, a terminal tool should not respond to Connection Manager messages.

Keep your tool self-contained

From the user's perspective, installing communications tools should simply be a matter of dragging the icon for that tool into the Communications folder on their desktop. To achieve this level of simplicity, your tool should be totally self-contained; all the resources that it needs to properly operate should be in the resource bundle.

There are, however, two exceptions to this. The first is when your tool uses a specific hardware interface that requires a driver to be loaded at INIT time. This circumstance is unavoidable, so it is an allowable exception to this guideline. The second exception is when your tool provides access to special data files (for example, a file of network addresses) that are kept on the user's system. Such data files provide your tool with a convenient way to store and distribute configuration information. In a case like this, your tool should save in the session document all user settings; your tool should not require external files to reestablish a previously configured connection. Whenever your tool does require an external file to operate properly, it should check for the existence of that file and notify the user if the file is not present.

Keep your tool task-specific

Special K supports three kinds of communications tools: connection, terminal, and file transfer. Your tool should be one of these types and should not implement any services that another type of tool is intended to provide. For instance, if you are writing a terminal tool, it should not provide any connection services. This guideline helps insure that tools will not interact with each other in undesireable ways. The services that each type of tool is meant to provide are:

- For connections tools: control the data path and its specifications. This includes altering the data path as needed or stripping high bits.
- For terminal tools: control user input and output. This includes input from the moust or keyborad and output to the terminal emulation window.
- For file transfer tools: conrol disk files. Only file transfer tools should manipulate disk files or the hierarchical file system (HFS).

Tools written for Special K are meant to be used such that users can change one part of a communications configuration and still have the application work. For instance, a user running a VT100 terminal emulation over an XMODEM connection should be able to run the emulation over an X.25 connection and not notice any changes.

However, if a terminal or file transfer tool requires a specific type of connection (due to the protocol or standard being implemented) that is not in place, your tool should send an error back to the application. In fact, any tool you write that works only when operating with another specific tool should be able to detect the presence or absence of that tool and send back appropriate return codes to the application. A tool should never cause a system-level error when a user tries to use it in the "wrong" configuration.

When writing a tool to implement an existing communications standard, you might find that the functions included in the standard span more than one type of tool. In cases like this, try to keep your tool task-specific by making use of the Macintosh interface. For example, if a connection protocol requires that your tool have status information constantly available, your tool can display this information in a separate window. You can also implement the standard by writing two task-specific tools that must be used together.

User interface considerations

This section describes the user interface considerations you should keep in mind when designing your tool. The following topics are included in this section:

- Modeless tool operation
- Contents of the configuration dialog box
- Appearance of windows and status dialog boxes
- Using menus
- Error handling
- Using the right words

Modeless tool operation

Your tool should operate modelessly because the Communications Toolbox (and most applications that use it) allow for multiple simultaneous communications sessions; yours may not be the only session running (and your tool may be in use in more than one session at a time). Also keep in mind that even if the user is running a single session, he or she may be running under MultiFinder.

Although specific applications can present other user interfaces, the user will usually configure a tool from within an application using the Configure dialog box, open or close the connection with menu items, and send or receive files with menu items. These are the basic aspects of the user action interface.

The user will usually create a new document, configure it using the configuration dialog box, and save it. Your tool should save all user settings in the session file, typically in a separate resource for each of the communications tool types (connection, terminal, and file transfer). The design of Special K assumes that the application will save settings in session documents so that a user can use a preconfigured document to open a connection; a user who uses several setting combinations is expected to prepare and use a separate document for each combination.

User should not need to perform more configuration when they open a connection or transfer a file; the only dialog boxes that should appear at this time are status dialog boxes. Therefore, your tool should fill in appropriate default settings when it is first selected in the Configure dialog box.

The configuration dialog box

Since users can use different tools inside the same application, the configuration dialog boxfor each tool should be visually compatible with those of other tools. This allows users to transfer knowledge they have about configuring one type to tool to configuring another type of tool. The six tools that come with this realease of Special K implement the visual interface that your tool should look like.

This visual style is similar to the usual style for modal dialog boxes but has some extensions for communication tools. Since many communications tools require more user-settable parameters than can be displayed nicely in a modal dialog box the size of the Macintosh Plus screen, your tool should use 9-point Geneva for tool controls instead of 12-point Chicago, which it would normally use.

The tools included in this release of Special K use small graphics to provide feedback on user settings. These graphics sometimes change appearance in response to a setting change (for example, to visibly show the effect of inverse video). This type of graphic information can help the user determine the effect of a setting change. Such small pictures should not be controls; they should not respond to mouse clicks. Rather, they should change appearance (if appropriate) in response to the user making a setting change using the usual controls.

If your tool is complex and requires more controls than can fit in a modal dialog box, even using 9-point Geneva, it should divide these controls among two or more screens. Your tool should allow users to select which screen of controls to view by selecting an icon in a scrolling list at the left edge of the dialog box, in a manner similar to the Control Panel desk accessory. The controls should be grouped according to function. Your tool should place the controls a user is most likely to select in the first screen displayed when the configuration dialog box comes up; it should place"power user" controls in subsequent screens.

If your tool displays a control that is dependent (that is, is enabled or disabled depending on the setting of another item), your tool should place the control under or alongside its controlling item and the item should be grayed out when it is disabled.

Since the configuration dialog box is modal, your tool should not use additional modal dialog boxes that pop-up on top of the configuration dialog box. If your tool requires a cascading dialog box, it should be limited to selection dialog boxes like `SFGetFile`, which controls settings that do not usually need to be changed. Your tool should never have more than two layers of modal dialog boxes on the screen at the same time.

If a field allows only certain types of data input (for example, only numbers), your tool should perform error checking by intercepting and checking keystrokes. In general, your tool should alert the user as soon as he or she makes the error, rather than waiting until the screen is dismissed. This helps the user find where the error occured on complex dialog boxes.

Windows and status dialog boxes

The terminal window is the only window that any of the communications tools display during normal operation. However, since a connection or file transfer tool should not place text in the terminal window, these tools should display their own window or modeless dialog boxes.

Status dialog boxes are the most common way for tools to request input or display output. When a tool performs an operation that will take a long time—for example, transferring a file or establishing a complex connection—the tool should post a status dialog box. This status dialog box should:

- be modeless
- incorporate quantitative status information, such as a progress thermometer, when appropriate
- contain a Cancel button to allow the user to abort the operation. (The command-period method of cancellation be problematic because multiple sessions may be running; the user could inadvertantly cancel dialog boxes other than they one they intend to cancel by hitting the command-period key combination several times.)

A tool might also put up its own window for user input and output during a session. For example, a connection tool might provide a command window that allows users to type in commands directly to control the connection. Your tool should either display this kind of window when the application initially selects your tool, or install a custom menu item that toggles in a manner similar to Hide Clipboard/Show Clipboard. Keep in mind that all command functions should be available through standard Macintosh controls, such as menu items and configuration dialog box settings. If your tool displays a command-line mode for compatibility with an existing standard, your tool should supplement the standard Macintosh interface rather than replacing it.

Menus

Your tool can place a menu of its own in the menu bar of the application. However, it should avoid doing this because the menu bar has limited available space and application designers tend to assume they can use the entire menu bar. Also, since up to three tools can be active at once, up to three tool menus might be displayed in addition to the menus owned by the application. Due to the potential space problems, your tool should avoid displaying menus. If you do choose to implement a menu for your tool, choose a menu name that is as short as practical to avoid overflowing the menu bar.

Tool-specific menus are placed to the right of application menus. This means that if the menu items of your tool have command-key equivalents, they will override any conflicting command keys for application menus. If there two tool menus displayed at the same time, the menu farthest to the right will override the other in a similar fashion. Furthermore, applications that allow scripted selection of menu items will select the rightmost menu item in case of a name conflict. Keep these potential sources of conflict in mind when designating names and command-key equivalents for a tool menu.

Your tool can have hierarchical menus. However, because hierarchical menus take more time for users to navigate through, your tool should use this kind of menu only for items that users do not often select. Communications tools can use tearoff menus where appropriate for the function being implemented(for example, for a keypad menu) and when supported by the System software.

Handling errors

Your tool should allow users to setup any communications configuration, even those that are unusable. This allows a system administrator to configure and save a session document for another person, who uses a configuration different from that on the administrator's machine. In cases like this, your tool should return an error only if the user attempts to open a connection, start a terminal emulation, or initiate a file transfer using an unusable setup.

Using the right words

Macintosh developers normally use terms that are intuitive and easy to learn, even for naïve users. However, this practice sometimes conflicts with the need to use established industry standard terms, which may be difficult for the novice to understand. Since communication software developers often implement pre-existing industry standards, this problem is especially common for developers of communication tools.

Where standard terms for a function already exist and are widely accepted, you should use the standard terms. This is both to ensure that your tool properly implements the standard and that experienced communication users who are familiar with the standard terms are not confused. However, you should attempt to make these terms as easily understandable as possible for inexperienced users. You can do this in several ways; alternate standard terms are sometimes available (for example, the terms *Show Controls* and *Transparent Mode* are used for the same VT102 terminal setting). You might also be able to embed the standard term in a longer description or use small images to make meanings more clear.

Compatibility Requirements

Tools should be compatible with all configurations with which Special K is compatible. Special K requirements are:

- Macintosh Plus (128K) ROMs or later
- 1M of RAM minimum
- System 6.02 and later

In order to be compatible with future releases of system software, it is important that your tool be 32-bit clean.

Terminal tool considerations

Terminal tools should support all Macintosh keyboards, including the original Macintosh keyboard (with and without detachable keypad), the Macintosh Plus keyboard, as well as the Standard, Extended, and Apple IIgs ADB keyboards. If arrow keys, function keys, the control key, or other keys are required by your tool but are not on the keyboard, your tool should provide an alternative means of accessing them (your tool could provide a keypad menu or allow the user to use the command key as a control key.

Appendix B **Useful Code Samples**

About this appendix

This appendix shows you solutions to common programming problems:

- Implementing effective idle loops
- Determining events that need to be handled by one of the Special K managers
- Customizing the tool-selection dialog.
- Determining if the Special K managers are installed
- Finding the `procID` of a tool
- Finding a `toolID`

Using `FTExec` and `TMIdle` effectively

The following code sample shows when your application needs to call `FTExec` and `TMIdle` during a file transfer.

```
PROCEDURE DoIdle;
VAR
      theWindow     : WindowPtr;
      theData             : WindowP;

      status              : LONGINT;
      sizes               : BufferSizes;
      theErr              : CMErr;
      flags               : INTEGER;
      mouseLoc      : Point;
      oldLeft             : INTEGER;
      oldBottom     : INTEGER;
      hScroll,
      vScroll             : ControlHandle;

      doFT          : BOOLEAN;
      doTM          : BOOLEAN;

      savedPort     : GrafPtr;
      tempString    : STR255;
      theReply      : SFReply;
      toprow              : INTEGER ;

BEGIN
      IF pigMode THEN
            DoMissPiggy;

      GetPort(savedPort);
      theWindow := FrontWindow;
      WHILE theWindow <> NIL DO
```

```
BEGIN
SetPort(theWindow);
theTerm  := GethTerm(theWindow);
theConn  := GethConn(theWindow);
theFT    := GethFT(theWindow);

IF theConn <> NIL THEN
       BEGIN
       CMIdle(theConn);
       theErr  := CMStatus(theConn, sizes, status);
       END; {theConn <> NIL }

doFT  := FALSE;
doTM  := TRUE;

IF theFT <> NIL THEN
       BEGIN
       theData  := WindowP(GetWRefCon(theWindow));

       IF BAND(theFT^^.flags, FTIsFTMode) <> 0 THEN
              BEGIN
              IF NOT theData^.wasFTMode THEN
                     BEGIN
                     theData^.wasFTMode  := TRUE;
                     DirtyMenu  := TRUE;
                     END;

              doFT  := TRUE;
              IF BAND(theFT^^.attributes, FTSameCircuit) <> 0 THEN
                     doTM  := FALSE;
              END
       ELSE
              BEGIN
              IF theData^.wasFTMode THEN
                     BEGIN
                     dirtyMenu  := TRUE;
                     theData^.wasFTMode  := FALSE;

                     IF BAND(theFT^^.flags, FTSucc) <> 0 THEN
                            BEGIN
                            END
                     ELSE
                            BEGIN
                            NumToString(theFT^^.errCode,
                            tempString);
                            IF theFT^^.errCode = 0 THEN
                                   DoErrorAlert(errorFTAborted, '')
                            ELSE
                                   DoErrorAlert(errorFTFailed,
                                   tempString);
                            END;
              { need to check if the AutoRec has anything in it }
                     tempString  := theFT^^.autoRec;
                     IF (theData^.searchBlk = 0) AND (tempString <>
```

```
                        '') THEN
                              theData^.SearchBlk :=
                              CMAddSearch(theConn, tempString, 0,
                              @AutoRecCallback);
                        IF theData^.searchBlk = -1 THEN
                        BEGIN
                              DebugStr('cannot add search');
                              theData^.searchBlk := 0;
                        END;
                        END;
                  END;

                  IF theData^.startFT THEN
                  BEGIN
                        theConn := GethConn(theWindow);
                        IF theConn <> NIL THEN
                              IF theFT^^.autoRec <> '' THEN
                              BEGIN
                                    IF theData^.searchBlk <> 0 THEN
                                          CMRemoveSearch(theConn,
                                          theData^.searchBlk);
                                    theData^.searchBlk := 0;
                              END;

                        theData^.startFT := FALSE;
                        theReply.vRefNum := 0;
                        theReply.fName := '';
                        theErr := FTStart(theFT,FTReceiving,theReply);
                  END;
            END;

      IF doFT THEN
            BEGIN
            IF theFT <> NIL THEN
                  BEGIN
                  FTExec(theFT);
                  END;
            END;

      IF theTerm <> Nil THEN BEGIN
            {Set cursor to arrow if outside the active tool}
            if (theWindow = FrontWindow) then begin
                  GetMouse(mouseLoc);
                  if NOT PtInRect(mouseLoc,theTerm^^.viewRect) THEN
                        Initcursor;

                  {Set the scrol bar value correctly}
                  { Put in idle loop because we're never sure when
}
                  { the tool will scroll
                  }
                  hScroll := GetHScroll(theWindow);
                  vScroll := GetVScroll(theWindow);
                  if (hScroll <> nil) and (vScroll <> nil) then begin
```

```
                                    oldLeft := GetCtlValue(hScroll);
                                    oldBottom := GetCtlValue(vScroll);
                                    toprow := theTerm^^.visRect.top-1 +
                                    theData^.cachelinecount;
                                    if (theTerm^^.visRect.left <> oldLeft) then
                                    SetCtlValue(hScroll,theTerm^^.visRect.left);
                              {     if (theTerm^^.visRect.bottom <> oldBottom)
                                          then
                              SetCtlValue(vScroll,theTerm^^.visRect.bottom);}
                        { we only want to adjust the scrollvalue according to the }
                                    { visRect.top if now cached line is visible }
                        if ( theTerm^^.viewRect.top = theTerm^^.termRect.top ) then
                                          if (  toprow <> oldBottom) then
                                                SetCtlValue(vScroll,toprow );
                              end;
                        end;

                              if doTM then
                                    TMIdle(theTerm);
                  END;

                  IF doTM THEN
                        BEGIN
                        IF theConn <> NIL THEN
                              BEGIN
                              IF BAND(status, CMStatusOpen + CMStatusDataAvail) <>
                              0 THEN
                                    BEGIN
                                    IF sizes[CMDataIn] <> 0 THEN
                                          BEGIN {sizes <> 0}
                                          IF sizes[CMDataIn] > myBufferSize THEN
                                                sizes[CMDataIn] := myBufferSize;
                                          theErr := CMRead(theConn, myBuffer,
                                          sizes[CMDataIn], CMData,
                                                FALSE, NIL, 0, flags);

                                          IF theData^.startFT THEN
      { we triggered a file transfer }
                                          BEGIN
                                                sizes[CMDataIn] :=
LONGINT(theData^.position) - LONGINT(myBuffer) - LENGTH(theFT^^.autoRec);
                                                IF sizes[CMDataIn] > 0 THEN
                                                      sizes[CMDataIn] :=
TMStream(theTerm, myBuffer, sizes[CMDataIn], flags);
                                          END
                                          ELSE
                                          BEGIN
                                                sizes[CMDataIn] :=
TMStream(theTerm, myBuffer, sizes[CMDataIn], flags);
                                          END;

                                          END; {sizes <> 0}
                                    END; {status oen }
                              END; {conn<> NIL}
```

```
                END;

            theWindow  :=  WindowPtr(WindowPeek(theWindow)^.nextWindow);
            END;

    SetPort(savedPort);
END;
```

Determining events for Special K managers

The following three routines show how an application can determine if an event needs to be handled by one of the File Transfer Manager event processing routines. If you are not writing an application that uses the File Transfer Manager routines, you can still learn from this example because it deals with events and windows—concepts that are common to all Macintosh programming.

The first routine, `IsFTWindow`, determines whether or not a window belongs to the File Transfer Manager. Windows (or dialogs) that belong to the File Transfer Manager should have a connection handle stored in the `refCon` field of the `windowRecord`.

The second routine, `IsFTEvent`, takes an event record and determines whether or not `FTEvent` should be called.

The third routine demonstrates calling `IsFTEvent` in the main event loop of an application.

```
FUNCTION      IsFTWindow(theWindow: WindowPtr): BOOLEAN;
VAR
      pWindow: WindowPtr;
      tempFT: FTHandle;
      hFT: FTHandle;

BEGIN
      IsFTWindow := FALSE;

      IF WindowPeek(theWindow)^.windowKind <> dialogKind THEN
            Exit(IsFTWindow);

      tempFT := FTHandle(GetWRefCon(theWindow));

      pWindow := FrontWindow;

      WHILE pWindow <> NIL DO
            BEGIN
            hFT := GethFT(pWindow);
            IF hFT <> NIL THEN
                  BEGIN
                  IF LONGINT(hFT) = LONGINT(tempFT) THEN
                        BEGIN
                        IsFTWindow := TRUE;
                        Exit(IsFTWindow);
                        END;
                  END;
            pWindow := WindowPtr(WindowPeek(pWindow)^.nextWindow);
            END;
END;


FUNCTION IsFTEvent(theEvent: EventRecord): FTHandle;
VAR
      theWindow : WindowPtr;
```

```
        hFT: FTHandle;

BEGIN
        IsFTEvent := NIL;
        theWindow := NIL;

        CASE theEvent.what OF
                autoKey, keyDown:    { no command-key equivalents on a mac
plus }
                        BEGIN
                        theWindow := FrontWindow;
                        END;
                mouseDown:
                        BEGIN
                        IF FindWindow(theEvent.where, theWindow)=0 THEN
                                ;
                        END;
                updateEvt:
                        BEGIN
                        theWindow := WindowPtr(theEvent.message);
                        END;
                activateEvt:
                        BEGIN
                        theWindow := WindowPtr(theEvent.message);
                        END;
                END; {case}

        IF theWindow <> NIL THEN
                BEGIN
                IF IsFTWindow(theWindow) THEN
                        BEGIN
                        hFT := FTHandle(GetWRefCon(theWindow));
                        IsFTEvent := hFT;
                        END
                ELSE
                        BEGIN
                        hFT := GethFT(theWindow);
                        IF hFT <> NIL THEN
                                BEGIN
                                IF BAND(hFT^^.flags, FTIsFTMode) <> 0 THEN
                                        IF BAND(hFT^^.attributes,
                                                FTSameCircuit) <> 0 THEN
                                                IF theEvent.what IN
                                                [autoKey, keyDown] THEN
                                                IsFTEvent := hFT;
                                END;
                        END;
                END;
END;

{$S EventSeg}
FUNCTION IsConnEvent(theEvent: EventRecord): ConnHandle;
VAR
        theWindow : WindowPtr;
```

```
        hConn: ConnHandle;

BEGIN
        IsConnEvent := NIL;
        theWindow := NIL;

        CASE theEvent.what OF
                autoKey, keyDown:    { no command-key equivalents on a mac
plus }
                        BEGIN
                        theWindow := FrontWindow;
                        END;
                mouseDown:
                        BEGIN
                        IF FindWindow(theEvent.where, theWindow)=0 THEN
                                ;
                        END;
                updateEvt:
                        BEGIN
                        theWindow := WindowPtr(theEvent.message);
                        END;
                activateEvt:
                        BEGIN
                        theWindow := WindowPtr(theEvent.message);
                        END;
                END; {case}

        IF theWindow <> NIL THEN
                BEGIN
                IF IsConnWindow(theWindow) THEN
                        BEGIN
                        hConn := ConnHandle(GetWRefCon(theWindow));
                        IsConnEvent := hConn;
                        END;
                END;
END;

{$S EventSeg}
FUNCTION IsTermEvent(theEvent: EventRecord): TermHandle;
VAR
        theWindow : WindowPtr;
        hTerm: TermHandle;

BEGIN
        IsTermEvent := NIL;
        theWindow := NIL;

        CASE theEvent.what OF
                autoKey, keyDown:    { no command-key equivalents on a mac
plus }
                        BEGIN
                        theWindow := FrontWindow;
                        END;
                mouseDown:
```

```
                BEGIN
                IF FindWindow(theEvent.where, theWindow)=0 THEN
                        ;
                END;
            updateEvt:
                BEGIN
                theWindow := WindowPtr(theEvent.message);
                END;
            activateEvt:
                BEGIN
                theWindow := WindowPtr(theEvent.message);
                END;
            END; {case}

        IF theWindow <> NIL THEN
            BEGIN
            IF IsTermWindow(theWindow) THEN
                BEGIN
                hTerm := TermHandle(GetWRefCon(theWindow));
                IsTermEvent := hTerm;
                END;
            END;
END;


PROCEDURE MainLoop;
VAR
        theEvent : EventRecord;
        theWindow : WindowPtr;
        theWindowPeek : WindowPeek;
        theControl : ControlHandle;
        savedPort : GrafPtr;
        theKey : CHAR;

        processed : BOOLEAN;
        result : LONGINT;
        hFT: FTHandle;

BEGIN
        WHILE NOT done DO
            BEGIN
            SystemTask;

            DoIdle;                     { application idle loop
procedure }
            IF WaitNextEvent(everyEvent,theEvent, 0, NIL) THEN
                BEGIN
                hFT := IsFTEvent(theEvent);
                IF hFT <> NIL THEN
                        FTEvent(hFT, theEvent)
                ELSE
                        BEGIN
                        CASE theEvent.what OF
                                autoKey, keyDown:
```

```
                        DoKey(theEvent);
                            mouseDown:
                                DoClick(theEvent);
                            updateEvt:
                                DoUpdate(theEvent);
                            app4Evt:
                                DoResume(theEvent);
                            activateEvt:
                                DoActivate(theEvent);
                            END; {case}
                        END;
                    END; {gne}
                END; {if done}
END;
```

Custom tool-selection dialog boxes

The sample code that follows shows how an application can use Connection Manager routines to present the user with a custom dialog box that can be used to select or customize a tool. This code calls a total of six Connection Manager routines..

CHOOSE.A

```
CallIdle    PROC    EXPORT

             MOVEM.L D0-D7/A0-A6,-(SP)
            MOVE.L  64(SP),A0
            JSR     (A0)
             MOVEM.L (SP)+,D0-D7/A0-A6
            RTS

            END
```

CHOOSE.P

```
UNIT Choose;
{

     This unit performs the standard dialog for configuration and
     selection of a communications tool.
}
INTERFACE

USES
     MemTypes, QuickDraw, OSIntf, ToolIntf, PackIntf,
     SpecialUnit,
     CTBUtils,
     TMIntf,
     CMIntf,
     FTIntf;

FUNCTION ChooseEntry(VAR theHandle:ConnHandle;
                                 where : Point; idleProc:ProcPtr) : INTEGER;

IMPLEMENTATION
{$SETC debugging:=FALSE}

CONST
     ChooseItemOK = 1;
     ChooseItemCancel = 2;
     ChooseItemOutline = 3;
     ChooseItemTitle = 4;
     ChooseItemVersion = 5;
     ChooseItemDottedLine = 6;
     ChooseItemPopup = 7;
```

```
TYPE
    dialogInfoP = ^dialogInfo;        { storate private to the config dialog }
    dialogInfo = RECORD
        tempProcID   :   INTEGER;     { MUST be the first item in record }
        magicCookie  :   LONGINT;     { MUST be the second item in the record }

        tempConfig   :   Ptr;         { config record being used }
        count        :   INTEGER;
        title        :   STR255;      { tool being displayed }
    END;

{   forward declaration }
FUNCTION     DoNewConn(VAR hConn:ConnHandle; tempProcID:INTEGER;
                       tempConfig:Ptr): BOOLEAN; FORWARD;

PROCEDURE    DrawLine(theDialog : DialogPtr; itemNo : INTEGER); FORWARD;
PROCEDURE    DrawTitle(theDialog : DialogPtr; itemNo : INTEGER); FORWARD;

PROCEDURE    SetDTextInfo(theDialog : DialogPtr; procID: INTEGER); FORWARD;

PROCEDURE    CallIdle(idleProc:ProcPtr); EXTERNAL;

FUNCTION     ChooseFilter(theDialog : DialogPtr; VAR theEvent:EventRecord;
                 VAR theItem:INTEGER) : BOOLEAN; FORWARD;

{============================================================================}
{
    theHandle is a (var) parameter which is the connection handle.
    where is the upper left hand corner of the selection dialog.
    idleProc is a procedure pointer (with no parameters) that is
        to be called from the dialog idle loop.
}
FUNCTION     ChooseEntry(VAR theHandle: ConnHandle; where: Point;
                idleProc: ProcPtr): INTEGER;

VAR
    maxExtent    :   Rect;            { max size of dialog in global coordinates }

    savedPort    :   GrafPtr;         { saved port }
    theWindow    :   WindowPtr;       { for invalidating after DisposDialog }
    theDialog    :   DialogPtr;       { the choose dialog }
    theControl   :   ControlHandle;

    tempString   :   STR255;          { temporary placeholder }
    tempTool     :   STR63;           { currently selected tool name }
    oldName      :   STR63;           { initially selected tool name }

    theItem      :   INTEGER;         { for manipulating dialog items }
    itemKind     :   INTEGER;
    itemHandle   :   Handle;
    itemRect     :   Rect;

    oldSize      :   Point;           { old size of dialog before resizing }
```

```
    theType      :   ResType;          { resource type to work with }
    thePtr       :   Ptr;              { ptr to temporary configuration record }

    oldVal       :   INTEGER;          { old control (popup menu) value }
    newVal       :   INTEGER;          { current control value }

    configSize   :   LONGINT;          { size of the configuration record }

    infoP        :   dialogInfoP;      { pointer to dialog data }

    hTerm        :   TermHandle;       { temporary variables }

    hMenu        :   MenuHandle;       { handle to popup menu control's menu }

    hDITL        :   Handle;           { handle to DITL to append }

    theErr:      OSErr;                { for building list of tools }
    vRefNum:     INTEGER;
    dirID:       LONGINT;

    tempPort:    GrafPtr;              { temporary holding place }

    theContext: CRMToolContext;
    err: OSErr;

BEGIN
    ChooseEntry := ChooseFailed;        { pessimistic happening }
    IF BAND(idleProc, 1) <> 0 THEN      { check for negative idle proc case }
    BEGIN
        Exit(ChooseEntry);
    END;

    InitCursor;                                          { reset to arrow }
    GetPort(savedPort);                                  { always the boyscout }

    theDialog := GetNewDialog(chooseResourceBase, NIL, POINTER(-1));
    IF theDialog = NIL THEN                              { perform safe dialog }
        Exit(ChooseEntry);

    SetPort(theDialog);                                  { at 'em boy }
    maxExtent := theDialog^.portRect;                    { my, how BIG }

    infoP := dialogInfoP(NewPtr(SIZEOF(dialogInfo)));    { internal data space }
    IF infoP = NIL THEN                                  { no memory }
    BEGIN
        DisposDialog(theDialog);                         { clean up }
        SetPort(savedPort);                              { set the port back }
        Exit(ChooseEntry);                               { bye }
    END;

    SetWRefCon(theDialog, ORD4(infoP));                  { set the refcon to
                                                           infoP }
```

```
                                                    { get title of dialog
                                                      and set up title
                                                      drawing userItem }
 GetIndString(infoP^.title, chooseResourceBase, 1);
 GetDItem(theDialog, ChooseItemTitle, itemKind, itemHandle, itemRect);
 SetDItem(theDialog, ChooseItemTitle, itemKind, @DrawTitle, itemRect);

WITH infoP^ DO
    BEGIN
    count := CountDITL(theDialog);                  { # items in DITL }

      GetDItem(theDialog,ChooseItemDottedLine,itemKind,itemHandle,itemRect);
      SetDItem(theDialog,ChooseItemDottedLine,itemKind,@DrawLine,itemRect);

    tempProcID := theHandle^^.procID;               { get the tool procID }
    thePtr := theHandle^^.config;                   { get the config field }
    CMGetToolName(tempProcID, tempString);          { get the toolname }
    oldName := tempString;                          { save the toolname }
    IF oldName = '' THEN                            { I am John Doe }
    BEGIN
        DisposPtr(Ptr(infoP));                      { get rid of dlog data }
        DisposDialog(theDialog);                    { clean up }
        SetPort(savedPort);                         { set the port back }
        Exit(ChooseEntry);                          { bye }
    END;

    configSize := GetPtrSize(thePtr);               { get size of config record }
    IF MemError <> noErr THEN                       { memory problem }
    BEGIN
        DisposPtr(Ptr(infoP));                      { get rid of dialog data }
        DisposDialog(theDialog);                    { clean up }
        SetPort(savedPort);                         { set the port back }
        Exit(ChooseEntry);                          { bye }
    END;
    tempConfig := NewPtr(configSize);               { copy it if possible... }
    IF tempConfig = NIL THEN                        { didn't get it }
    BEGIN
        DisposPtr(Ptr(infoP));                      { get rid of dialog data }
        DisposDialog(theDialog);                    { clean up }
        SetPort(savedPort);                         { set the port back }
        Exit(ChooseEntry);                          { bye }
    END;
     BlockMove(thePtr, tempConfig, configSize);     { copy it }

                                                    { draw outline for
                                                      default button }
     GetDItem(theDialog, ChooseItemOutLine, itemKind, itemHandle, itemRect);
     SetDItem(theDialog, ChooseItemOutLine, itemKind,
         @GoodDrawOutlineButton, itemRect);

                                                    { set up popup menu }
     GetDItem(theDialog, ChooseItemPopup, itemKind, itemHandle, itemRect);
     theControl := GetNewControl(chooseResourceBase, theDialog);
    IF theControl = NIL THEN
```

```
BEGIN
     DisposPtr(Ptr(infoP));                    { get rid of dlog data }
     DisposDialog(theDialog);                  { clean up }
     SetPort(savedPort);                       { set the port back }
     Exit(ChooseEntry);                        { bye }
END;

 hMenu := GetMHandle(chooseResourceBase);
IF hMenu = NIL THEN
BEGIN
     DisposPtr(Ptr(infoP));                    { get rid of dlog data }
     DisposDialog(theDialog);                  { clean up }
     SetPort(savedPort);                       { set the port back }
     Exit(ChooseEntry);                        { bye }
END;

theItem := 1;                                  { index thru tools }
theErr := noErr;
WHILE theErr = noErr DO                        { while no problems }
BEGIN
      theErr := CRMGetIndToolName( 'cbnd', theItem, tempString);
     tempTool := tempString;

     IF theErr = noErr THEN                     { no problems officer }
     BEGIN
          IF tempTool <> '' THEN                { got one! }
          BEGIN
               AppendMenu(hMenu, 'X');          { this is to prevent }
                SetItem(hMenu, theItem, StringPtr(@tempTool)^);
                                                 { problems with special
                                                   menu chars like / etc }
               theItem := theItem + 1;          { get the next one please }
          END;
     END;
END; {while}

theItem := CountMItems(hMenu);                 { How many tools? }


tempString := oldName;                         { try to select the
                                                     appropriate item }
 oldVal := FindMenuItem(hMenu, tempString);    { in the popup menu }

IF oldval = 0 THEN                             { Add the current tool if
                                                     it's not already there }
BEGIN
{
 The user has moved the file out of the communications directory -
 we can show the name, but this menu item needs to be disabled
}
     theItem := theItem + 1;                    {Update these counts}
     oldval := theItem;
      AppendMenu(hMenu, 'X');
       SetItem(hMenu,theItem,StringPtr(@oldName)^);
```

```
        DisableItem(hMenu, theItem);{ disable it }
END;

IF theItem = 0 THEN
BEGIN
    DisposPtr(Ptr(infoP));                          { get rid of dlog data }
    DisposDialog(theDialog);                        { clean up }
    SetPort(savedPort);                             { set the port back }
    Exit(ChooseEntry);                              { bye }
END;
SetCtlMax(theControl, theItem);                     { we've added items so
                                                        set up max of ctl }

 GetIndString(tempString, chooseResourceBase, 2);{ get Popup Title }
SetCTitle(theControl, tempString);                  { set the title }

itemRect := theControl^^.contrlRect;                { fix rectangle size }
 SetDItem(theDialog, ChooseItemPopup, itemKind, itemHandle, itemRect);

oldSize := theDialog^.portRect.botRight;            { old size of dialog }
 UnionRect(maxExtent, theDialog^.portRect, maxExtent);
                                                    { grow max dlog size }

newVal := oldVal;
SetCtlValue(theControl, oldVal);                    { set up popup value }

 hDITL := CMSetupPreflight(tempProcID, magicCookie);
                                                    { get DITL to append }

 SetDTextInfo(theDialog,theHandle^^.procID);        { Set the dialog's text
                                                        info    }


 AppendDITL(theDialog, hDITL, appendDITLBottom); { append it }
IF hDITL <> NIL THEN
    DisposHandle(hDITL);                            { done with the DITL }

 CMSetupSetup(tempProcID, tempConfig, count+1, theDialog, magicCookie);
                                                    { set up the items }

 UnionRect(maxExtent, theDialog^.portRect, maxExtent);
                                                    { grow max dlog size }
                                                    { we need to do this
                                                        after every call to
                                                        appendditl }
 MoveWindow(theDialog, where.h, where.v, TRUE);  { move dialog  }
ShowWindow(theDialog);                              { unveiling… }

theItem := 0;                                       { for now… }
 WHILE (theItem <> ChooseItemOK) AND (theItem <> ChooseItemCancel) DO
    BEGIN
    IF idleProc <> NIL THEN                         { call the idle proc }
        CallIdle(idleProc);
```

```
ModalDialog(@ChooseFilter, theItem);          { modal dialog   }
IF theItem = ChooseItemPopup THEN             { did popup get hit? }
    BEGIN
     newVal := GetCtlValue(theControl);       { what is new value? }
    IF newVal <> oldVal THEN                  { it has changed! }
        BEGIN
          CMSetupCleanup(tempProcID, tempConfig, count+1,
             theDialog, magicCookie);
                                              { cleanup the setup   }
          ShortenDITL(theDialog, CountDITL(theDialog) - count);
          CMSetupPostflight(tempProcID);      { decrement usecount
                                                of tool }

          SizeWindow(theDialog, oldSize.h, oldSize.v, TRUE);
                                              { change dialog size }

          GetItem(hMenu, newVal, tempString);     { get new tool name }
          tempProcID := CMGetProcID(tempString); { get procID }
         DisposPtr(tempConfig);                   { get rid of
                                                       old config }
        tempConfig := NIL;                        { pessimistic }
         CMDefault(tempConfig, tempProcID, TRUE);
                                                  { and get a new one }

          hDITL := CMSetupPreflight(tempProcID, magicCookie);
                                                  { get a new DITL }
          SetDTextInfo(theDialog,tempProcID);
         IF hDITL <> NIL THEN                     { sanity chex }
        BEGIN
              AppendDITL(theDialog, hDITL, appendDITLBottom);
                                                  { append it }
            IF hDITL <> NIL THEN
                 DisposHandle(hDITL);             { append and then
                                                       get rid of it }
        END;

          CMSetupSetup(tempProcID, tempConfig, count+1, theDialog,
            magicCookie);
                                                { set up the items }
          SizeWindow(theDialog, theDialog^.portRect.right,
                                  theDialog^.portRect.bottom, FALSE);
                                                { size the window }
         oldVal := newVal;                      { update value to
                                                  detect toolchange }
          UnionRect(maxExtent, theDialog^.portRect, maxExtent);
                                                { grow max dlog size }
                                                { do this after every
                                                  appendditl }

        END;
    END; {item = count }

IF theItem > count THEN                         { tool's item hit }
      CMSetupItem(tempProcID, tempConfig, count+1, theDialog,
```

```
        theItem, magicCookie);
END; {while theItem NOT in 1..2}

HideWindow(theDialog);                          { hide the dialog }
newVal := GetCtlValue(theControl);              { check name change}
GetItem(hMenu, newVal, tempString);             { get the new name }
tempTool := tempString;                         { save the new name }
tempProcID := CMGetProcID(tempString);

{ this is to keep track of the maximum size of the dlog so that
    we can invalidate the proper areas of the windows behind
    when the dialog disappears
}
LocalToGlobal(maxExtent.topLeft);          { convert maxExtent }
LocalToGlobal(maxExtent.botRight);         { to global coordinates  }

                                           { spring cleaning }
CMSetupCleanup(tempProcID, tempConfig, count+1, theDialog,
   magicCookie);
ShortenDITL(theDialog, CountDITL(theDialog) - count);
CMSetupPostflight(tempProcID);

DisposDialog(theDialog);                   { bye bye dialog }
SetPort(savedPort);                        { don't leave port wierd }

IF theItem = ChooseItemOK THEN             {OK}
BEGIN                                        { has the name of tool
                                               changed?
                                               be case INsensitive,
                                               diacrit sensitive }
    IF NOT EqualString(oldName, tempTool, FALSE, TRUE) THEN
    BEGIN
        ChooseEntry := ChooseOKMajor;
        tempString := tempTool;
        tempProcID := CMGetProcID(tempString);

        IF NOT DoNewConn(ConnHandle(theHandle), tempProcID,
                tempConfig) THEN
            ChooseEntry := ChooseAborted;

        IF theHandle = NIL THEN                { disaster! }
        BEGIN
            ChooseEntry := ChooseDisaster;
        END
        ELSE
        BEGIN
            configSize := GetPtrSize(tempConfig);
            BlockMove(tempConfig, theHandle^^.config, configSize);

            IF CMValidate(theHandle) THEN { validate for kicks }
              ;
        END;
    END
    ELSE
```

```
                BEGIN                                        { same tool, so validate }
                     ChooseEntry := ChooseOKMinor;
                     configSize := GetPtrSize(tempConfig);
                      BlockMove(tempConfig, theHandle^^.config, configSize);
                     IF CMValidate(theHandle) THEN
                        ;
                END;
            END
            ELSE
            BEGIN                                       { use hit CANCEL }
                 ChooseEntry := ChooseCancel;
            END;


            DisposPtr(tempConfig);                          { done with config }
            DisposPtr(Ptr(infoP));                          { get rid of dlog data }

        { now we need to go through window list and update
             all areas that were ever covered up by the
             config dialog which has grown and potentially
             shrunk.  we have kept track of the largest size of
             the dialog.  we will now convert it to global coords
             and invalrect everybody in the window list
        }
             GetPort(savedPort);
             theWindow := FrontWindow;
             WHILE theWindow <> NIL DO
             BEGIN
                  SetPort(theWindow);
                  itemRect := maxExtent;
                  GlobalToLocal(itemRect.topLeft);          { get max extent in
                                                                local coords }
                  GlobalToLocal(itemRect.botRight);         { ditto }
                  InvalRect(itemRect);

                   theWindow := WindowPtr( WindowPeek(theWindow)^.nextWindow );
             END;
              SetPort(savedPort);
        END;      {with }
END;

{   this is to confirm shutting down the connection to modify it or not }
FUNCTION ReallyShutdown: BOOLEAN;
VAR
     theDialog: DialogPtr;
     theItem: INTEGER;
     savedPort: GrafPtr;

     itemKind     :    INTEGER;
     itemHandle   :    Handle;
     itemRect     :    Rect;

BEGIN
     ReallyShutdown := TRUE;                            { reckless }
```

```
    GetPort(savedPort);                                    { save the port }
     theDialog := GetNewDialog(ChooseResourceBase+1, NIL, WindowPtr(-1));
    IF theDialog = NIL THEN                                { no dialog }
         Exit(ReallyShutdown);
                                                           { set up outline button }
     GetDItem(theDialog, 3, itemKind, itemHandle, itemRect);
     SetDItem(theDialog, 3, itemKind, @GoodDrawOutlineButton, itemRect);

    CenterWindow(theDialog);                               { center it }
    SysBeep(5);
    ShowWindow(theDialog);                                 { show it }
    SysBeep(5);
     ModalDialog(NIL, theItem);
    ReallyShutdown := (theItem = 1);                       { 1 = ok, 2 = cancel }
    DisposDialog(theDialog);                               { rid o the dialog }
    SetPort(savedPort);                                    { restore the port }
END;

{    change from one connection type to another }
FUNCTION DoNewConn(VAR hConn:ConnHandle; tempProcID:INTEGER;
             tempConfig:Ptr): BOOLEAN;
VAR
     savedDesiredSizes    :    BufferSizes;
     savedRefCon          :    LONGINT;
     savedUserData        :    LONGINT;
     savedFlags           :    LONGINT;
     savedMLU             :    LONGINT;

     savedReserved0       :    LONGINT;
     savedReserved1       :    LONGINT;
     savedReserved2       :    LONGINT;

     status          : LONGINT;
     sizes           : BufferSizes;
     theErr          : CMErr;

BEGIN
     DoNewConn := TRUE;
     theErr := CMStatus(hConn, sizes, status);             { get conn status }
     IF theErr = noErr THEN                                { OK }
          IF BAnd(status, CMStatusOpen+CMStatusOpening) <> 0 THEN
          BEGIN                                            { conn currently open }
               IF NOT ReallyShutdown THEN                  { confirm this please }
               BEGIN                                       { no confirmation }
                    DoNewConn := FALSE;                    { so beat a quick
                                                             retreat }
                    Exit(DoNewConn);
               END;
          END;

     WITH hConn^^ DO      { save any desired parameters }
          BEGIN
          savedFlags := flags;
          savedDesiredSizes := BufSizes;
```

```
        savedRefCon := refcon;
        savedUserData := userData;
        savedMLU := MLUField;
        savedReserved0 := reserved0;
        savedReserved1 := reserved1;
        savedReserved2 := reserved2;
      END;

    CMDispose(hConn);                                   { get rid of old conn }

     hConn := CMNew(tempProcID, savedFlags, savedDesiredSizes, savedRefCon,
                savedUserData);
    IF hConn <> NIL THEN                                { sanity chex }
        WITH hConn^^ DO
        BEGIN
            MLUField := savedMLU;
            reserved0 := savedReserved0;
            reserved1 := savedReserved1;
            reserved2 := savedReserved2;
        END;
    DoNewConn := TRUE;

END;

{   Choose dialog filter procedure }
FUNCTION ChooseFilter(theDialog : DialogPtr; VAR theEvent:EventRecord;
                      VAR theItem:INTEGER) : BOOLEAN;
VAR
    theControl :    ControlHandle;
    where :         Point;

    result :        BOOLEAN;
    theKey :        CHAR;

    savedPort :     GrafPtr;
    theWindow :     WindowPtr;                          { for event processing }

    pDialogInfo :   DialogInfoP;                        { dialog private data }

    itemKind :      INTEGER;
    itemHandle :    Handle;
    itemRect :      Rect;

    theKeys:        KeyMap;

BEGIN
    theItem := 0;                                       { nothing initially }

    result := FALSE;                                    { for now… }

    IF theEvent.what = keyDown THEN
    BEGIN
        theKey := CHAR(BAND(theEvent.message, charCodeMask));
```

```
    IF (theKey = CHAR($03)) OR (theKey = CHAR($0D)) THEN
                                                    { enter or return }
    BEGIN
        theItem := ChooseItemOK;                    { OK button }
        result := TRUE;
    END;

    IF (theKey = '.') AND (BAND(theEvent.modifiers, cmdKey) <> 0) THEN
    BEGIN                                           { cmd- '.' }
        theItem := ChooseItemCancel;
        result := TRUE;
    END;

    IF result = TRUE THEN                { hilite if we got a button }
    BEGIN
        GetDItem(theDialog, theItem, itemKind, itemHandle, itemRect);
        HiliteControl(ControlHandle(itemHandle), 1);
    END;

    IF result THEN                        { we have preprocessed the
                                            RETURN, ENTER,
                                            and cmd-. }
    BEGIN
        ChooseFilter := TRUE;
        Exit(ChooseFilter);
    END;
END;


pDialogInfo := DialogInfoP(GetWRefCon(theDialog));   { get the dlog data }
WITH pDialogInfo^ DO
BEGIN
    result := CMSetupFilter(tempProcID, tempConfig, count+1, theDialog,
                theEvent, theItem, magicCookie);

    ChooseFilter := result;                          { TRUE or FALSE }
    IF result THEN                                   { it WAS processed }
        Exit(ChooseFilter);                          { so exit }
END;

result := FALSE;                                     { pessimism }
CASE theEvent.what OF                                { process the event }
    updateEvt:
    BEGIN
        GetPort(savedPort);                          { get the port }
        theWindow := WindowPtr(theEvent.message);    { get the update owner }
        SetPort(theWindow);
        BeginUpdate(theWindow);
        EraseRect(theWindow^.portRect);              { erase }

        IF theWindow = theDialog THEN                { process if ours }
            UpdtDialog(theDialog, theWindow^.visRgn);

        EndUpdate(theWindow);                        { otherwise eat it }
```

```
        SetPort(savedPort);
    result := TRUE;                                  { chomp chomp }
END;
activateEvt:
BEGIN
     theWindow := WindowPtr(theEvent.message);    { eat the activates }
     IF BAND(theEvent.modifiers, activeFlag) <> 0 THEN
         IF theWindow = theDialog THEN
             SetPort(theWindow);                     { set port if activate }
    result := TRUE;
END;
mouseDown:
BEGIN
     where := theEvent.where;                  { where was the mousedown }
     GlobalToLocal(where);                     { convert to local coords }
                                               { did we click in control? }
     IF FindControl(where, theDialog, theControl) <> 0 THEN
    BEGIN
          IF TrackControl(theControl, where, POINTER(-1)) <> 0 THEN
                                                { so track it }
        BEGIN
            result := TRUE;                     { we got the event }
             theItem := FindDItem(theDialog, where) + 1;
                                                { so item hit }
        END
        ELSE
        BEGIN                                    { tracked out of it }
            result := TRUE;
            theItem := 0;                       { so no item hit }
        END;
    END;
END;
keyDown:                                        { keyDown }
BEGIN
     theKey := CHAR(BAND(theEvent.message, charCodeMask));

     IF (theKey = CHAR($03)) OR (theKey = CHAR($0D)) THEN
                                                { enter or return }
    BEGIN
        theItem := ChooseItemOK;                { OK button }
        result := TRUE;
    END;

     IF (theKey = '.') AND (BAND(theEvent.modifiers, cmdKey) <> 0) THEN
    BEGIN                                      { cmd- '.' }
        theItem := ChooseItemCancel;
        result := TRUE;
    END;

    IF result = TRUE THEN                       { hilite if we got a button }
    BEGIN
         GetDItem(theDialog, theItem, itemKind, itemHandle, itemRect);
         HiliteControl(ControlHandle(itemHandle), 1);
    END;
```

```
        END;
        otherwise
        BEGIN
        END;
    END; {case}
    ChooseFilter := result;
END;


{   draw title of the user item }
PROCEDURE DrawTitle(theDialog : DialogPtr; itemNo : INTEGER);
VAR
    infoP           :   dialogInfoP;

    itemHandle      :   Handle;
    itemRect        :   Rect;
    itemType        :   INTEGER;
    itemKind        :   INTEGER;

    savedFont       :   INTEGER;
    savedSize       :   INTEGER;

BEGIN
     infoP := DialogInfoP(GetWRefCon(theDialog));
    WITH infoP^ DO
        BEGIN
         savedFont := theDialog^.txFont;
         savedSize := theDialog^.txSize;

        TextFont(0);                    { system font and size please }
        TextSize(0);

         GetDItem(theDialog, itemNo, itemKind, itemHandle, itemRect);
        EraseRect(itemRect);            { erase it please }
         TextBox( Ptr( ORD4(@title) + 1), LENGTH(title), itemRect, teJustLeft);

         TextFont(savedFont);           { restore font and size information }
         TextSize(savedSize);
        END;
END;

{   useritem to draw dotted line }
PROCEDURE DrawLine(theDialog : DialogPtr; itemNo : INTEGER);
VAR
    itemHandle      :   Handle;
    itemRect        :   Rect;
    itemType        :   INTEGER;
    itemKind        :   INTEGER;
    thePattern      :   Pattern;
    savedPen        :   PenState;

BEGIN
    GetIndPattern(thePattern, sysPatListID, 4);     { gray pattern }
```

```
     GetPenState(savedPen);
     PenNormal;

{    PenPat(thePattern); }

     GetDItem(theDialog, itemNo, itemKind, itemHandle, itemRect);
     FrameRect(itemRect);

     SetPenState(savedPen);
END;

{    this routine will set the dialog font and size
     information based upon the finf resource
}
PROCEDURE SetDTextInfo(theDialog: DialogPtr; procID: INTEGER);
VAR
     savedPort        : GrafPtr;
     thefinf          : finfRecord;
     myPeek           : DialogPeek;
     savedState       : SignedByte;
     info             : FontInfo;
     curResRef        : INTEGER;
     theResource      : Handle;

BEGIN
     GetPort(savedPort);                                  { safe porting }
     SetPort(theDialog);

     curResRef := CurResFile;
     UseResFile(procID);              { look for finf in tool first }

     theResource := Get1Resource('finf',chooseResourceBase);
     if (theResource <> nil) then begin
          GetIndfinf(@thefinf,chooseResourceBase,Window_Info);
                                   { Get the indexed info from finf resource }
          UseResFile(curResRef);
     end
     else begin
          UseResFile(curResRef);
          GetIndfinf(@thefinf,chooseResourceBase,Window_Info);
     end;

     myPeek := DialogPeek(theDialog);

     savedState := HGetState(Handle(myPeek^.textH));      { get handle state }
     HLock(Handle(myPeek^.textH));                        { get the TE handle }

     with thefinf do
     begin

          TextSize(theSize);                          { Load in the grafport info }
          TextFont(fontNum);
          TextFace(theFace);
```

```
        with myPeek^.textH^^ do
        begin                                          { Load in the TE info     }
            txFont := fontNum;
            txFace := theFace;
            txSize := theSize;

            WITH info DO
            BEGIN                                      { stuff fields in TERecord }
                 lineHeight := ascent + descent + leading;

                fontAscent := ascent;                          { I-378 }
            END;
        end;

    end;

    HSetState(Handle(myPeek^.textH), savedState);      { restore dialog }
    SetPort(savedPort);                                { and now back to our
                                                           regular station }
END; { SetDTextInfo }


END.
```

CHOOSE.R

```
#include "SysTypes.r"
#include "Types.r"

#define ChooseResourceBase   256
#define Popup   32                          /* 2 * 16 */

/* Font Info for dialog items    */
resource 'finf' (ChooseResourceBase, purgeable) {
     {    /* array Fonts: 2 elements */
          /* [1] */                         /* Window Font  */
          3,                                          /* font number */
          plain,                                           /* style */
          9,                                               /* size
          /* [2] */                         /* Icon Font    */
          3,
          plain,
          9
     }
};

resource 'STR#' (ChooseResourceBase, "Titles for config dialog") {
     {
      "Connection Configuration",
     "Method:",
     ".",
     }
};

resource 'DLOG' (ChooseResourceBase, "setup dialog") {
     {0, 0, 70, 450},
     dBoxProc,
     invisible,
     noGoAway,
     0x0,
      ChooseResourceBase,
     "Setup Dialog Box"
};

resource 'DLOG' (ChooseResourceBase+1, "confirm closing connection") {
     {0, 0, 100, 300},
     dBoxProc,
     invisible,
     noGoAway,
     0x0,
      ChooseResourceBase+1,
     "confirmation dialog"
};

resource 'CNTL' (ChooseResourceBase, "Tools control ") {
     {30, 5, 50, 300},
     -1,              /* right just       */
```

```
    visible,
    90,                 /* width of title */
    ChooseResourceBase,             /* menu associated */
    Popup,              /* no options CDEF 10 = 16 * 10 + variation code */
    0,                   /* reference menu 11000, popup title width 50 */
    "title:"
};
resource 'DITL' (ChooseResourceBase, "Basic configuration DITL") {
    {    /* array DITLarray: 5 elements */
        /* [1] */
        {32, 370, 52, 440},
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {5, 370, 25, 440},
        Button {
            enabled,
            "Cancel"
        },
        /* [3] outline of OK button */
        {28, 366, 56, 444},
/*      {35, 370, 55, 440},      */
        UserItem {
            enabled
        },
        /* [4] title */
        {5, 5, 21, 200},
        userItem {
            disabled
        },
        /* [5] version */
        {50, 85, 60, 130},
        UserItem {
            disabled
        },
        /* [6] dotten line separating static from dynamic portions */
        {62, 0, 63, 9999},
        UserItem {
            disabled
        },
        /* [7] select tool popup menu user item */
        {30, 5, 50, 300},
        UserItem {
            enabled
        }
    }
};

resource 'DITL' (ChooseResourceBase+1, "Confirmation DITL") {
    {    /* array DITLarray: 5 elements */
        /* [1] */
        {70, 220, 90, 290},
```

```
        Button {
            enabled,
            "OK"
        },
        /* [2] */
        {40, 220, 60, 290},
        Button {
            enabled,
            "Cancel"
        },
        /* [3] outline of OK button */
        {66, 216, 94, 294},
        UserItem {
            enabled
        },
        /* [4] text */
        {10, 5, 90, 200},
        StaticText {
            disabled,
             "Modifying the current connection may cause the "
                    "connection to close.   Proceed?"
        }
    }
};

resource 'MENU' (ChooseResourceBase, "Popup Menu") {
     ChooseResourceBase,
     textMenuProc,
     allEnabled,
     enabled,
     "Choose Menu",
     {
     }
};
```

Determining if the managers are installed

```
FUNCTION Installed : BOOLEAN;
CONST
     CommToolboxTrap = $8B;
     UnimplementedTrapNumber = $9F;

BEGIN
     Installed := TRUE;
      IF NGetTrapAddress(UnimplementedTrapNumber, OSTrap) =
                                          NGetTrapAddress(CommToolboxTrap, OSTrap)
THEN
       BEGIN
         Installed := FALSE;
       END;
END;
```

Getting the procID

```
FUNCTION GetProcID(msg: integer; name : str255) : Integer;
BEGIN
       CASE (msg) OF
              NEWCM:
                     GetProcID := CMGetProcID(name);
              NEWTM:
                     GetProcID := TMGetProcID(name);
              NEWFT:
                     GetProcID := FTGetProcID(name);
       END;

END;
```

Finding the tool ID

```
FUNCTION FindToolID(msg : integer): integer;
VAR
      theType:      ResType;
      err:          OSErr;
      tempTool:     STR255;
      procID:             Integer;

BEGIN
      FindToolID := -1;                { nothing doing }
      procID := -1;

      GetIndString(tempTool,DEFAULTSTR,msg+1);
      {Try to get the default}
      if LENGTH(tempTool) > 0 then
            procID := GetProcID(msg, tempTool);
            {Get the ProcID if possible}

      if procID = -1 then
      { The default tool wasn't specified or doesn't exist}
      BEGIN
            CASE (msg) of
                  NEWCM:
                        theType := ClassCM;
                  NEWTM:
                        theType := ClassTM;
                  NEWFT:
                        theType   := ClassFT;
            END;

            err := CRMGetIndToolName(theType, 1, tempTool);

            if err <> noErr then
            BEGIN
                  DebugStr('FindTool - no tools');
                  EXIT(FindToolID);
            END;

            procID := GetProcID(msg, tempTool);

            IF procID = -1 THEN
            BEGIN
                  DebugStr('CTBGetProcID returned -1');
            END;
      END;
      FindToolID := procID;
END;
```