Protocol Mover

AN EXTERNAL PROTOCOL INTERFACE

VERSION 0.60

Introduction:

The files that are included with this documentation show an example external for ASCII (non-protocol) transfers. ASCII and the documentation that follows will get you started. There will probably be a lot of questions this first draft does not answer. If you have a question about the protocol interface you can find me at: 76174,205 on Compuserve and D3885 on AppleLink.

The Interface:

When starting either an receive or a send the terminal or BBS program will call the external with a newly created ProtoRec data structure. This will be done until the external returns with a non-zero value. The external will defined as:

```
    pascal OsErr protocol(message, PRP, refCon)
    int message;
    ProtoRecPtr PRP;
```

long refCon;

message will be either SEND, RECEIVE or SETPREFS on the first call. On jump backs the message is should not be used and can be ignored by the external.

PRP is a pointer to a ProtoRec structure set up by the communications program for the external to use. On the first call it will contain initial values set by application.

refCon is the value setup in the 'PInf' resource for accessing the protocol. It may be used for setting protocol options or for selecting different protocols that the external supports.

Possible return values are standard Mac OS errors with the addition of 128 = transfer has either aborted or has finished.

The new ProtoRec variables created by the communications application are shown below. Initial values are in braces {value}.

mRefIn {Modem input port reference number for current node}
mRefOut {Modem output port reference number for current node}

procID {resource ID of the protocol}

This is used by the protocol to reference any other resources it may need. All PROCs should be compiled with a resource ID of 1000. Any other resources your protocol might need should be number from 1000 to 1099. The Protocol Mover will renumber all resources as necessary. To find a resource the external should add this number to a base value. For example: If you have a dialog with an ID of 1000 you should reference it as 0+procID.

protoData{0L}

To be used as a handle for the external's private data. It should be initialized on the first call and disposed of when the external returns an error or abort/finish code.

errReason {0L}

Is a short pascal string that describes the last error condition. This will be used on the fly in the communications program transfer dialog for non-fatal errors and in the transfer log for fatal errors. The easiest way to set this variable is with a NewString call. newError must be set to TRUE in order for the communications program to display and dispose of the message.

timeOut {timeOut}

Is the maximum time the external should wait before returning to communications program.

This value may change while the transfer is occurring depending upon system load.

fileCount {Total number of files to transfer}

For batch receives in terminal mode this will always be set to one.

filesDone {0}

After each successful transfer the external will add one to this count until the transfer is aborted or until filesDone = fileCount. The exception to this rule is batch receives in terminal mode. In this case filesDone should remain zero.

bytesDone {0L}

Number of bytes successfully transfered. Updated continuously by the external.

bytesTotal {0L}

Total number of bytes transfered by the external.

startTime {TickCount()}

Time of transfer start.

The following is set up as a bit field:

transMode {mode}

Set to either TERMINALMODE or BBSMODE. Depending upon the environment the external may have to act in different ways. For example: If it is being used to receive files in the communications program terminal mode MacBinary files should be

renamed to whatever MacBinary file name they have. If it is being used in the transfer section communications program will supply the file name. transMode will tell the external what it is being used for.

stopTrans{FALSE}

If this value becomes TRUE the external should abort the transfer as soon as possible.

carrierLost {FALSE}

If this field becomes true you can assume that the connection lost carrier. In this case you should stop sending and receiving data through the current port.

useMacBinary {useMacBinary}

If this is false MacBinary will not be used at all. Otherwise smart MacBinary detection should be used by the external.

newMBName {FALSE}

When receiving a MacBinary file the external should place the MacBinary name in mbName and set newMBName to TRUE.

newError {FALSE}

If this becomes true communications program should use errReason to update the transfer dialog. After doing so communications program should dispose of errReason, set it to 0L, and set newError to FALSE. **Hint:** If using StackSaver it is wise to immediately call Return with a zero timeOut value after setting newError. This would force a return to communications program which would show the errReason string and clear newError.

newFile {FALSE}

If this becomes true communications program should use the information contained in the structure to update everything for a successful transfer. After doing so communications

program should set newFile to FALSE. filesDone will be the index number of the successfully transfered file. **Hint:** If using StackSaver it is wise to immediately call Return with a zero timeOut value after setting newFile. This would force a return to communications program which would take care of updating all of the user stats for a successful transfer and clear newFile.

Recovering {FALSE}

The external should set this to TRUE if the external is either sending or receiving a only part of a file. This way the communications program can use the current value contained in bytesDone as a reference when computing cps.

Reserved {0}

Do not look at and do not change!

fList[] {the file to transfer}

The FListRec contains information about each file that has to be transfered.

fName {file path name}

The full or partial path name for the file to be transfered. fName along with mbName (if used) will be disposed of by communications program when the transfer is complete.

mbName {0L, or MacBinary name}

For BBS use only. When uploading a file might be saved under a name that is different from its “real” MacBinary name. Hence, the BBS program can use mbName to keep track of what the file should really be called. When downloading the opposite is true.

vRefNum {0, vRefNum}

The volume reference number for the file if is a full path name, or a working directory reference number in the case of partial path names.

DirID {0L}

The directory ID of the folder for the file.

FileID {0L}

Not supported yet.

IT CAN BE ASSUMED THAT THE PRP IS NOT A DEREFERNCED HANDLE THAT WOULD MOVE BETWEEN CALLS TO THE EXTERNAL DURING A TRANSFER. IT CAN ALSO BE ASSUMED THAT THE EXTERNAL WILL REMAIN LOCKED DOWN BETWEEN ALL CALLS TO IT!!! ALL STRING HANDLES SHOULD BE UNLOCKED!

THE STACK SAVER:

```
typedef struct {
	Handle stackData;
	long originalSP;
	long jumpBack;
	long startTime;
} SSaverRec, *SSaverPtr;

/*	prototypes for StackSaver */

extern	void	Return(int	timeOut,SSaverPtr
SSP);
```

```
extern OsErr JumpBack(int message,ProtoRecPtr prp,
                      long refCon,SSaverPtr SSP,OsErr procPtr());
```

stackData must be initialized to a handle of zero length on the first call to the external. When the external is completely finished with what it has to do it must dispose of stackData before returning with a non-zero error code.

The other variables are for StackSaver's internal use and should not be touched by the external.

When writing an external you may wish to observe the stack size by watching how large stackData becomes when returning to the test shell. This will give you some idea of the overhead involved when returning to communications program. For example: If the stackData size is about 256 bytes then that would be similar to two Str255 copies; One for returning, and one for coming back. Of course

it would be wise to keep this number as small as possible. If you have a large number of local variables in the stack frame when Return is called you might want to try moving Return elsewhere or possibly using variables located in protoData rather than on the stack.

StackSaver is presently written for a 'C' environment. If you need StackSaver for pascal please leave me E-Mail. We can work something out in no time if you are serious about creating a protocol external in pascal.

GLOBAL WARNING!:

THINK C allows the use of global variables within code resources. When running in a multi-node environment and especially when using StackSaver great care must be taken with global variables.

Since the global variables will be re-initialized *every time the external is called* they should be thought of as temporary storage places at best.

Any variable that you want to keep around between JumpBacks *must be kept in the protoData handle of the ProtoRec structure!*

Where do I start? (from the application side):

If you are writing a terminal program you will have to do very little coding yourself. sample.c is part of Protocol Mover and is provided as an example of how to interface with the interface.

utility.c is actually the source to a special PROC that is around all of the time in the Transfer Protocols file. The calling method for your application is shown in sample.c. The utility PROC does not have to be called until it returns an error. It can be assumed that it will finish whatever you tell it to do on the first call.

Besides the calls to the utility PROC shown in sample.c it is also called by the Protocol Mover as shown below:

To do a Transfer menu command…

```
MenuID   = HiWord(MenuSelection);
MenuItem = LoWord(MenuSelection);
switch(MenuID) {
case 0:
      break;
case AppleID:
      DoAppleCmd(MenuItem);
      break;
case FileID:
      DoFileCmd(MenuItem);
      break;
case EditID:
      DoEditCmd(MenuItem);
      break;
case PortID:
      DoPortCmd(MenuItem);
      break;
case DataRateID:
      DoDataRateCmd(MenuItem);
      break;
case BitsID:
      DoBitsCmd(MenuItem);
      break;
case FlowControlID:
      DoFlowControlCmd(MenuItem);
      break;
default:
      Utility(DOMENU, PMH, MenuSelection);
      showTdialog();
      break;
}
```

Notice we do a showTdialog right after the DOMENU call. The code for showTdialog is found in sample.c.

When maintaining the Transfer menus…

```
Utility(ABLEMENU, PMH, (long) (thePrefs.currentPort != -1));
```

When building the Transfer menu…

```
Utility(BUILDTMENU, &PMH,
            (((long) TransferID) << 16)+thePrefs.mode);
```

Note that the Protocol Mover uses a special BUILDTMENU message. This call will give you a full menu in BBSMODE. Your BBS application (and terminal program) should use BUILDMENU. For a BBS application this will only give you the Set and Help commands.

When disposing of the Transfer menu…

```
Utility(DISPOSEMENU, PMH, 0L);
PMH = 0L;
CloseResFile(rRef);
```

In addition to the Utility calls the other code in sample.c is called from the main event loop…

```
for (;;) {
      doTransPROC();
      if (gHasWaitNextEvent) {
            if (!WaitNextEvent(everyEvent, &myEvent, 0L, 0L)) {
                  continue;
            }
      }
      else {
            SystemTask();
            if (!GetNextEvent(everyEvent, &myEvent)) {
                  continue;
            }
      }
      MaintainCursor();
      if (IsDialogEvent(&myEvent)) {
            doTdialog(&myEvent);
      }
```

For a terminal program that is just about all you have to do. If you are writing a BBS program you will have to do a lot more coding on your own. The best place to start is the utility.c source. Examine and understand what happens when to send a BUILDTMENU call to it. This will help you with writing the code that lists the transfer protocols that are available to your users. Next you will want to understand what happens when you send a DOMENU call. For setting transfer prefs and getting help in BBS mode you can still use the Utility PROC, but when it comes to creating the file lists for either uploading or downloading you will have to do that yourself.

Utility Call Summary:

message BUILDTMENU, BUILDMENU and BUILDDAMENU

parameter1 address of the ProcMenuHandle

parameter2 long integer (menu ID in hi-word, transfer mode in low-word)

You pass the utility PROC the address of a variable that you want to hold the ProcMenuHandle and it returns with all of the protocol data filled out in the handle and the transfer menu items added to the menu. The long integer defines the ID of the Transfer menu. Note that this menu can have items on it already! The protocol functions will be added on to the end of the existing menu. The mode specifies either TERMINALMODE or BBSMODE. BUILDDAMENU is identical to BUILDMENU except desk accessory hierarchical menu IDs are.

message ABLEMENU

parameter1 the ProcMenuHandle

parameter2 long integer (TRUE or FALSE)

Given an existing ProcMenuHandle created by BUILDMENU this call will either enable or disable the transfer menu items.

message DOMENU

parameter1 the ProcMenuHandle

parameter2 the menu selection

Given an existing ProcMenuHandle created by BUILDMENU this call will take a menu selection long word and find the protocol function to execute. It will handle Set Receive Folder and Help calls in their entirety. For actual transfers it will set up the transIndex, transMessage, transRefCon, and transData fields of the ProcMenuHandle. It will be up to the application to check if transData was actually created and to take care of the transfer itself.

message DISPOSEPREC

parameter1 the ProcMenuHandle

parameter2 zero (long integer)

When a transfer is finished this call is used to dispose of the transData created by DOMENU.

message DISPOSEMENU

parameter1 the ProcMenuHandle

parameter2 zero (long integer)

When the transfer menu items are no longer needed this call will dispose of both the items in the menu and the ProcMenuHandle. Note that this call only disposes the transfer items in the menu and not the actual menu itself.

Notes:

Why have a transfer menu in BBS mode? So you can set the BBS preferences for the protocols that need them and still have the help menu available. You see, each protocol that handles prefs calls will keep two sets of prefs in a DATA resource. One for the Terminal Mode and the other for the BBS Mode. (See the ASCII example) This way someone could change the prefs when he is in Terminal Mode and not worry about effecting the operation of his BBS system. It would of course also be nice to have the help for all of the protocols available in BBS mode....it is a quick, easy, and cheap method of checking versions, author's addresses, shareware costs, etc.

Well, you will pretty much be building your own transfer menu for your BBS. How you want to handle that is up to you. See what happens in utility.c when it builds a

Terminal Mode menu....concentrate on the two calls to AddProcSubs when they are called with the mask of CANSEN+CANBSEN for sending and CANREC+CANBREC for receiving. This goes through all of the PInf resources and picks out the protocols that it can use. hCount is used to count the functions related to **the same PROC ID**. If it's greater than one I need a hierarchical menu, otherwise I can get away without one.

Then you can look at what happens on a DOMENU call. This searches through the structure built by BUILDMENU to find a matching PInf for the menu item.

There will be some work to do on your side, but when it's done you will have a killer transfer protocol system.

You need BUILDMENU even in BBS mode. You are right in that the user will never see it....that's why the transfer protocols (send and receive) will not be listed, but the **SYSOP** needs to set protocol prefs for BBS mode and get information about each protocol.