NetWork Transport Systems

NetWork Processor allows to add additional transport systems, up to a total number of four transport systems simultaneously. The maximum of four is a compile time constant of NetWork Processor – if you want to use/implement more than four, drop us a note. This document is intended to give you the necessary information to write new transport systems. It also provides some background information about the built-in transport systems “Dispatcher”, “Local”, and “AppleTalk”. The “Dispatcher” is special: it is used to select one of the other transport systems to send a message on, depending upon the destination address.

Transport systems may be implemented as code resources of type 'NetT' (id range 0..3), which are copied into NetWork Processor and loaded at system startup, or they can be implemented as applications, drivers, or any other piece of code and register themselves with the NetWork Processor. There is a slight difference between these two techniques which is documented below. It is probably best, to implement new transport systems by writing an application which implements the protocol, and move this application to the “:NetWork Startup Tools:” folder.

Transport System Interface

NetWork Processor uses the following record to keep information about transport systems. The record declaration is available in NetWork.p:

```
TransportRecord = record
        TransportProc   : Ptr;    { pointer to definition proc }
        TransportID     : longint;    { transport (unique) signature }
        TransportAddr : longint;    { local address of this transport system }
        TransportBCAddr       : longint;      { this transports broadcast address }
        TransportStart  : longint;    { first valid address }
        TransportEnd    : longint;    { last valid address }
        TransportMsgSize      : integer;      { size of MsgRecord for this transport system }
        TransportNumListens : integer;      { Number of listeners that should be active }
        TransportName: StringHandle;        { name of resource }
        TransportMsgQHead   : ^MsgPtr;    { pointer to head of queue }
        TransportVars   : Ptr;    { private vars, may be longint, ptr, or handle }
end;
TransportPtr = ^TransportRecord;
```

`TransportProc` is the entry of the procedure implementing the transport system. The procedure must be declared as follows (pascal calling conventions apply) :

procedure Transport (cmd: integer; Msg: MsgPtr; control: TransportPtr);

A single procedure is used to implement the entire transport system. The parameter `cmd` is used to tell the transport system which function to perform, see below. `Msg` is the message to operate on – if any –, and `control` points to this transport´s `TransportRecord`.

`TransportID` is your unique signature that you register with Apple. The use of a signature allows applications to differentiate among transport systems if necessary. E.g. the library on the NetWork distribution disk looks for the "AppleTalk" transport system. Other applications might want to communicate with some unix server, and therefore want to verify that a TCP/IP transport system is available.

`TransportAddr` is the node´s local address. Because NetWork Processor supports multiple transport systems, it does not assume a single local address, but matches against all the local addresses in all transport records. The "Dispatcher" is responsible to replace references to the local machine with the transport address of the transport system it selects to send a message.

`TransportBCAddr` – if not zero – is the broadcast or multicast address of this transport system. If `TransportBCAddr` is zero, then this transport system is not assumed to support broadcasts.

`TransportStart` and `TransportEnd` define the range of legal addresses for this transport system. The two numbers are interpreted as unsigned longs. All addresses that are valid for this transport system (including the local address) are assumed to be within this range, except for the broadcast address, which may be separate. The two numbers are used by "Dispatcher" (see below) to select a transport system.

`TransportMsgSize` is the size of a message. Transport systems that are implemented as 'NetT' resources and loaded at system startup can set `TransportMsgSize` to a value larger than the default `sizeof (MsgRec)`, in order to allocate additional space for private variables. Transport systems loaded after system startup cannot ask NetWork Processor to enlarge the size of a message. Instead, they can allocate their own memory (preferably from a private heap zone) and store a pointer to it in `MsgXFerID` (which is never looked at by NetWork itself).

NetWork Processor will try to allocate `TransportNumListens` listeners dedicated to this transport system. A listener is a `MsgRec`, that NetWork Processor allocates and passes to the transport system to store a received message in. The value of `TransportNumListens` can be changed at any time, though NetWork Processor will never abort listeners in excess of this value. E.g., the "AppleTalk" transport clears this value and aborts all

listeners, if AppleTalk is turned off. It sets `TransportNumListens` to 3, if AppleTalk is turned on again. The "Local" transport system sets this value to 1 if a message has been posted locally, and resets it to 0 if no more messages are pending.

`TransportName` is a `StringHandle` to the name of the transport system. If the transport system has been loaded at system startup, NetWork Processor takes care of storing the name of the resource. If you implement a new transport system, you should store a `StringHandle` in this field.

`TransportMsgQHead` contains a pointer to the head of the queue of active messages. This may be necessary if you want to look for a specific message in the queue. In general you will have to match the `MsgProtIndex` (the queue contains all active messages regardless of the transport system which is processing them). E.g., the "Local" transport uses this to match a listen with a post.

`TransportVars` is for use of the transport system. With the exception of transport system 0, NetWork Processor does never access this field. You can store a handle or a pointer in this field. You can even store the information directly into the field if it does not exceed four bytes. If you implement a new transport system within an application, it is probably best to store a reference to your global data (your A5) in this field.

Startup Operation

At system startup, NetWork Processor looks in its own resource file for resources of type 'NetT' with the ids 0 through 3. If it locates them, it initializes a `TransportRecord` by storing a pointer to the resource in `TransportProc` and copying the bytes 4 through 27 of the resource to the fields `TransportID` through `TransportNumListens`. After that it calls the transport system with the command `tInit` (see below for a complete reference of the commands). The transport system can change any field of the record it wants. After the call, NetWork Processor initializes the field `TransportMsgQHead`.

Note that your resource is assumed to start with a long branch, because NetWork Processor expects bytes 4..27 to contain data. See below for a sample transport resource.

NetWork Processor – as distributed – contains three built in transport systems. "Local" ('NetT' 1) and "AppleTalk" ('NetT' 2) are used to transfer messages locally or using AppleTalk, respectively. "Dispatcher" ('NetT' 0) is special. It is used to select a transport system to send a message on (unless the programmer specified otherwise). Its operation is detailed below.

In order to support "Dispatcher", NetWork Processor does two special things after it has loaded all transport systems

that are part of its resource file. It sets the “Dispatcher´s” `TransportMsgSize` to the maximum that has been requested by any transport system loaded at that time, and it sets `TransportVars` to the maximum transport system index that is supported by NetWork Processor.

The following table lists the initial values of the built-in transport systems:

TransportName	“Dispatcher”	“Local”	“AppleTalk”
TransportID	'NetD'	'NetX'	'NetA'
TransportAddr	0	0	net,node,socket (40)
TransportBCAddr	0	0	$FFFFFFFF
TransportStart	0	0	$00000101
TransportEnd	0	0	$FFFEFEFE
TransportNumListens		0	0 3

Note: A transport system (except “Dispatcher”) should not assume anything about other transport systems or the total number of transport systems. A transport system may however compare the value of `MsgProtIndex` of one message to the value stored in other messages, in order to find out which messages are also handled by this transport system.

“Dispatcher” Operation

If a message is posted, the library allocates a new `MsgRec` by calling NetWork Processor (see the library source for details). NetWork Processor clears the entire `MsgRec` before returning a pointer to it. Unless the programmer changes the value stored in `MsgProtIndex`, the “Dispatcher” (transport system 0) will be called during the actual post operation. “Dispatcher” supports exactly one command: `tPost`, all other commands are either ignored or cause an error condition.

“Dispatcher” looks at the destination address (actually `MsgDest.a`) and searches for a transport system that supports this address, either as the braodcast address or as a node address in the range defined by the transport system. If there is a transport system supporting the destination address, “Dispatcher” changes `MsgProtIndex` and also updates all references to the local node (`MsgSource.a`, `MsgDest.a`, and `MsgReply.a`) to be the local address of this transport system. If no transport system can be located, an error is returned.

Note that NetWork Processor saved the current `cmd` in `MsgCmd` and set the `MsgResult` to 1, indicating busy. “Dispatcher” does not change them, which causes NetWork Processor to restart the operation, this time calling a

different transport system.

Transport System Commands

NetWork Processor uses the following command codes to tell the transport systems, which operation to perform. All commands consist of a major and a minor command code. A list follows:

```
{ major command codes }
    tGeneral        = $00;
    tListen  = $10;
    tGet     = $20;
    tAccept = $30;
    tNew     = $40;          { used internally }
    tPost    = $50;
{ minor command codes }
    tStart   = $00;
    tAbort   = $0F;
{ misc command codes }
    tInit    = $00;
    tTickle  = $01;
    tRegister       = $04;
    tDeRegister     = $05;
{ useful values }
    tMajorMask     = $F0;
    tMinorMask     = $0F;
```

Commands are of two categories: general or message oriented. Let us first consider message transfer commands, which are in the range $10 through $5F.

Message Transfer Commands

The use of major and minor command codes provides an easy way to sequence through a series of steps, in order to perform one operation. If NetWork Processor wants to start a message transfer, say a post, it uses the command `tPost+tStart`, if it wants to abort a posted message, it calls `tPost+tAbort`. All other minor command values, range 1..14 (decimal), can be used for sequencing purposes. E.g., the "AppleTalk" transport system increments `MsgCmd` by one after it has started the operation. The next time it gets called with the same message, it knows that it must test, if the transfer has been completed.

Another feature of NetWork Processor also supports this type of sequencing: The transport system will be called repeatedly with the same message (regardless of the major command code) by calling it with the command stored in `MsgCmd`, as long as `MsgResult` is greater than zero. Note that a zero implies sucessful completion, whereas negative values are error indications. Btw, NetWork Processor considers it an error to start a new operation on a message as long as MsgResult is greater than zero.

The command `tListen` is used after a listener has been allocated (up to the number given by `TransportNumListens`, see above). A transport system should prepare to receive a new message, possibly allocating the memory required to do that. Note that you cannot use the memory pointed to by `sgPrioPtr` and `MsgStdPtr`, because this is application supplied memory after a get or accept has been performed. In fact, these fields are nil.

Listen (note that this can be any time after the initial call) should set `MsgResult` to zero, if a new message arrived. Also you must fill in the following fields of the message record: `MsgSource`, `MsgDest`, `MsgReply`, `MsgCapas`, `MsgIDStamp`, `MsgPrioSize`, and `MsgStdSize`. If an error is detected or if it is impossible to receive messages, return an error. Note that you should clear `TransportNumListens` if it is (temporarily) impossible to receive messages.

The command `tGet` is used during a GetMsg operation. It should be used to copy the priority information from your private memory to the memory pointed to by `MsgPrioPtr`. Note that you must never transfer more than `MsgPrioSize` bytes. Warning: though it is possible to sequence through a series of steps, NetWork Processor will block until the Get has completed.

The command `tAccept` is used during a AcceptMsg operation. It should be used to copy the standard information from anywhere to the memory pointed to by `MsgStdPtr`. Note that you must never transfer more than `MsgStdSize` bytes. NetWork does not require this command to complete immediately – it may take some time e.g. to transfer the information through a network connection.

The command `tNew` is used internally to mark a newly allocated message. A transport system will never receive this command code.

The command `tPost` is used to start a message transfer. All information required has already been filled into the

message record.

Any of the command codes above incremented by `tAbort`, implies that you should abort any transfer in progress and deallocate any additional memory you allocated. This command is used by NetWork Processor in one of two cases: It is used when the user program calls DestroyMsg and whenever a message reaches the timeout (usually two minutes). In the latter case, the command is actually called twice.

Note: It is probably best not to allocate memory in the system or application heaps in order to avoid fragmentation. Also, applications might not be prepared to handle moving blocks as a result of calling NetWork Processor.

General Commands

The following command codes do not operate on a message. NetWork Processor uses these command codes to perform special actions. All of them have a major command code of `tGeneral`, therefore only the minor command code is given. Also, none of these command can return an error.

The command `tInit` is called only once at system startup as documented above. The command `tTickle` is called periodically (somewhere between 10 and 20 times/second) and allows the transport system to perform housekeeping. E.g. “AppleTalk” uses this command to test if AppleTalk has been reopened and updates `TransportNumListens` accordingly.

The commands `tRegister` and `tDeRegister` are called if the system´s state changes from busy to idle or vice versa, respectively. “AppleTalk” uses this to change the NBP type from “NetWork Processor” to “NetWork User” or vice versa. If no such action is required, the command is to be ignored.

Sample Transport System

The following section lists the “Local” transport system. The assembly portion is listed first, followed by the pascal code.

```
; Copyright 1990 Joachim Lindenberg,
Karlsruhe

;

; this file implements the header of
the local transport procedure

;

   main

   import   Local

   bra  Local



   string asis



   dc.l 'NetX'   ; transport signature

   dc.l 0   ; local address of this
transport system

   dc.l 0   ;     this     transports
broadcast address

   dc.l 0   ; first valid address

   dc.l 0   ; last valid address

   dc.w 0   ; size of MsgRecord for
this transport system

   dc.w 0   ; Number of listeners that
should be active



   end
```

```
{ Copyright 1990 The NetWork Project,
StatLab Heidelberg,

   Copyright 1990 Joachim Lindenberg,
Karlsruhe. All rights reserved. }



unit LocalTransport;



interface



uses Types, Memory, NetWork;



procedure Local (cmd : integer; Msg :
MsgPtr; control : TransportPtr);



implementation



const        tPostMatched   =   tPost+1;
   { listen matched post }

         tPostDone = tPost+2;
   { accepted }



function   Min   (a,   b   :   longint)   :
longint;

begin

   if a < b then Min := a

   else Min := b;

end;



procedure MatchPost (var q : MsgPtr);
```

```
var p : MsgPtr;

begin

   p := q;

   while   (p   <>   nil)   and   not
( { complex matching.. }

             (p^.MsgProtIndex         =
protindex) & (p^.MsgCmd = tPost)

             & (p^.MsgResult > 0) ) do

         p := p^.MsgLink;

   q := p;

end;



procedure Local (cmd : integer; Msg :
MsgPtr; control : TransportPtr);

var result : integer; p : MsgPtr;

begin

   with Msg^ do begin

         if (Msg <> nil) & (BAnd (cmd,
tAbort) = tAbort) then begin

             result := eAbortMsg;

             if  MsgXFerID  <>  0  then
begin

                 MsgPtr
(MsgXFerID)^.MsgResult := eInvalid;

             end;

         end

         else  result  :=  0;  {  catches
all undefined cmds }



         case cmd of
```

```
            tGeneral                :
control^.TransportMsgSize :=

                sizeof (MsgRec);

            tListen : begin

                p                   :=
control^.TransportMsgQHead^;  MatchPost
(p);

                if   p   =   nil   then
result := eAbortMsg

                else begin

                  MsgSource         :=
p^.MsgSource;

                  MsgDest           :=
p^.MsgDest;

                  MsgReply          :=
p^.MsgReply;

                  MsgCapas          :=
p^.MsgCapas;

                  MsgStamp          :=
p^.MsgStamp;

                  MsgPrioSize       :=
p^.MsgPrioSize;

                  MsgStdSize        :=
p^.MsgStdSize;

                  MsgXFerID         :=
longint (p);

                  MsgPtr
(p)^.MsgXFerID := longint (Msg);

                  MsgPtr
(p)^.MsgCmd := tPostMatched;

                  { returns 0 }

                  MatchPost        (p,
Msg^.MsgProtIndex);

                    {    search    for
```

```
additional local messages }

                end;


   control^.TransportNumListens := ord
(p <> nil);

            end;



            tGet : begin

                BlockMove      (MsgPtr
(MsgXFerID)^.MsgPrioPtr,

                  MsgPrioPtr,      Min
(MsgPrioSize,

                    MsgPtr
(MsgXFerID)^.MsgPrioSize));

            end;



            tAccept : begin

                BlockMove      (MsgPtr
(MsgXFerID)^.MsgStdPtr,

                  MsgStdPtr,       Min
(MsgStdSize,

                    MsgPtr
(MsgXFerID)^.MsgStdSize));

                MsgPtr
(MsgXFerID)^.MsgResult := 0;

                MsgPtr
(MsgXFerID)^.MsgXFerID := 0;

                MsgXFerID := 0;

            end;



            tPost : begin
```

```
                MsgXFerID     :=     0;
result := 1;  { always 1 }


   control^.TransportNumListens := 1;

                   {     allocate     a
listener }

             end;

             tPostMatched  :  result  :=
1;

   {         tPostDone  :  ;  {  returns
0 }



         end; { case }


         if Msg <> nil then begin

             MsgResult := result;

         end;

   end;

end;


end.
```

1 a zero implies not supported.

2 that´s why “NetWork Communications” documents -1 as the broadcast address. If you wants to broadcast on all transport systems, you will have to index through all transport systems and post a message for each transport system separately unless the broadcast address is zero.