Center for Advanced Computation

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

CAC Document No. 246

A Network Unix System
Vol. 4: COINS Transponder Implementation

by
Richard Balocca

April 1978

CAC Document Number 246

A Network UNIX System
Volume Four: COINS Transponder Implementation

by

Richard Balocca

Prepared for the
Department of Defense

Center for Advanced Computation and
Computing Services Office of the
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

April 1978

Approved for release: Karl C Kelley
Karl.C. Kelley, Principal Investigator

Table of Contents

1 Document Overview

1.1 Overview

This document consist of three major parts: a User Manual, a Maintenance Manual, and an Implementation Manual. The User Manual describes the operation and control of the Transponder from the operator's view. The Maintenance Manual describes how to install the Transponder and where to find the various functional parts of it. It includes a set of source listings, indexed for convenience. The Implementation Manual describes the principles of Transponder operation.

2 Transponder User Manual

2.1 Introduction

The Unix Transponder is a test program for the Unix Network Control Program (NCP). The Unix Transponder can be used in conjunction with the Elf "COINS" Transponder, [1] which has a very similar the user and network interface. In fact the Elf COINS Transponder Program Description can be used as background to this document.

The Network Unix system is designed to operate with an Arpanet type of network, such as Platform or COINS, as well as the original Arpanet. The only differences involve user level programs. The Unix Transponder, although designed for the COINS network, will operate under any of these networks.

The purpose of the Unix Transponder is to provide an evaluation measure for system acceptance. It allows test data patterns to be sent and received with a wide dynamic range of demand rates. The data is sent to a host (possibly the same host) that is running a passive echo program known as a Responder. The echoed data is returned to the Transponder, where it is checked for accuracy. Two series of statistics are collected. These involve the timing and size of Network messages sent and received.

The Unix Transponder, unlike the Elf Transponder, does not require a stand-alone system. This has several advantages 1) It allows a testing of the true production system and NCP. Under Unix the majority of the NCP is a user level daemon controlled by messages. This fact, combined with the fact that every user level process has a separate address space, means that the Transponder has no special relationship with the NCP (that an ordinary user process does not have). 2) Unix users need not be affected during Transponder testing. 3) Multiple Transponders may be run, allowing the upper limit on the test load to be Unix' system limits. 4) The full power of the system can be used to examine the operation of the Transponder, the status of the Network, and the status of Unix.

2.2 Controlling the Transponder

The user issues commands which are interpreted by the Transponder. There are commands for creating, deleting (liquidating), and controlling network connections (links). The command may, depending on the command, affect zero or more links.

2.3 Link Status

The Transponder keeps track of the status of each of the links, along with the time at which the status last changed. There are eight possible status values, five of which correspond to steady states of the link (and are printed on the user's teleprinter as acknowledgements) and the other three of which are intermediate status values (never printed as acknowledgements), indicating that another state

----

[1] Which tests the Elf NCP, naturally.

change is imminent. The status information is kept in a status table.

Since the status is kept not only for timing purposes but also for verification purposes, the Transponder will print an interpretation of the acknowledgement on the user's teleprinter as it is occurs. The printed acknowledgements are presented on their own line, preceded by a left angle bracket and followed by a right angle bracket. They contain the link number and the type of acknowledgement. The type can be one of purged, ready, active, closed, ready-wait, active-wait, purge-wait, stop-wait, unused, or ERROR. The acknowledgement will appear on the user's teleprinter in the format:

<L ack>

where L is the link number in decimal, and ack is the particular status type. The status type will be surrounded by angle brackets in this document also. This is to distinguish them from commands, which will be surrounded with double quotes. The intermediate states <ready-wait>, <purge-wait>, <unused>, and <stop-wait> will never be printed as acknowledgements, but will appear in the status table (with the further exception of <unused>* which will not even appear in the status table). If an expected acknowledgement is not forthcoming, the Transponder is not functioning correctly. [2]

2.4 Commands

The Transponder is invoked from Unix with the command "master". There are no parameters to this command. At this point the user will receive a prompt and a herald with the date. The user may then type various commands to control the Transponder. A script of the user's commands along with certain responses such as printed acknowledgements are appended to a file with the name transponderlog in the current working directory. (These responses are Master responses--see the Transponder Implementation Manual for further information on what constitutes a Master response.)

Because of the nature of the Transponder, [3] the users should wait to type a command until the prompt appears. Also for the same reason, at least one line may be lost after the Transponder has exited.

The available Transponder commands are:

".",
"!",
"help",
"date",
"time",

----

[2] Check, for instance, to see that all requisite programs are present. Most of these program should be in the current working directory. Unless otherwise noted--i.e. if the filename is given by an absolute path name (one starting with a "/")--all files mentioned in this document will be relative to the current working directory.

[3] Specifically the use of an Echo Master and Command Master connected by pipes--See the Transponder Maintenance Manual.

```
"quit",
"pause",
"listen",
"ready",
"start",
"status",
"stop",
"purge",
"kill",
"i" or "indirect",
"p" or "param".
```

The commands are of the form

```
command firstarg, secondarg, ...
```

where the arguments are, depending on the command, either decimal numbers representing link numbers (a number assigned by the Transponder), octal numbers, or arbitrary strings. Most commands expect decimal link numbers. The "param" command, however, accepts all of its arguments in unsigned octal. This is for compatibility with the Elf Transponder, as well as permitting message patterns to be specified easily. [4]

2.4.1 Miscellaneous Commands

Prefixing a line with a colon (":") causes the rest of the line to be ignored.

Prefixing a line with an exclamation point ("!") will cause the Transponder to hand the line (minus the exclamation point) to the Unix shell. The interface between the user's terminal and the shell will involve a pipe, so the user should be aware that 'raw' mode and control-d will not work as expected. [5]

----

[4] It is an easily remedied deficiency of the Unix Transponder that decimal and octal numbers are not allowed everywhere, distinguished by local syntax.

[5] The Echo Master should be the process to interpret the "!" command even though currently it interprets no commands. The reason for this is that the Echo Master could pass the user's keyboard to the sub-shell. Then all commands would work as expected. The current implementation causes certain programs (those that expect to be able to set raw mode, for instance) to behave unexpectedly. Also no end of file, interrupt, or signal can be signaled from a pipe, thus the user must be careful not to execute programs that use those 'out-of-band' signals for information. For instance, cdb should not be run from the Transponder as the only way to exit from cdb is with an end of file! If the user gets him/herself into such an unfortunate dead-end situation, he/she can extradite him/herself by using the quit signal. This will kill the Echo and Command Master. It will generate a core image, but it will work.

The command "help" produces a list of all the commands with the count and format of their parameters. [6]

The commands "date" and "time" print Unix' idea of the current date and time respectively. Note that in contrast to the Elf Transponder, the Unix Transponder has no commands for setting these values. It is assumed they were supplied at bootstrap time.

The command "quit" causes the Transponder to do just that. The Transponder will make sure that all links are sent to the <unused> state before quiting. The network connections on any link in the <active> state, will be closed immediately, possibly in mid-message,

"Pause" followed by a decimal number will cause the Transponder to stop accepting commands and interpreting acknowledgements for that many seconds.

If the user types "listen", a Responder [7] process is created on the local host. It will listen on Network socket seven, which is a defacto Arpanet Standard for such echo daemons. This is in contrast with the Elf Transponder, which uses socket thirteen. [8]

2.4.2 Commands affecting Link Status

"Ready" has one parameter: a single link number corresponding to a <closed> link. The corresponding link will be put in the <ready> state (restarted) via the <ready-wait> state. A <ready> acknowledgement should be expected. The following is the state diagram for the ready command: [9]

----

[6] There are sharp signs in the help printout. They indicate that what is required in this position is an unsigned octal number.

[7] See below for a description of the purpose of the Responder.

[8] An additional way to start the Responder is with the Unix command "/etc/responder>/usr/lpd/responderlog&". The Responder statistics will appear in the file /usr/lpd/responderlog. Otherwise, there is no acknowledgement that the Responder is running. If it is desired that the Responder output be recorded somewhere else, it can be redirected to any file by naming the file to the right of a greater-than sign (">") as in: "/etc/responder>filename&".

[9] All commands which affect the state of a link will be diagrammed, just as the ready command is diagrammed here. The meaning of

A --stimulus--> B

is that if we are in state A and issue the indicated 'stimulus' then the link will enter state B. If B is followed by a star ("*") then the entry to state B is causes a printed acknowledgement to appear on the user's terminal.

```
<close> --"ready"--> <ready-wait>
<ready-wait> --internal processing delay--> <ready>*
```

In actuality not all the data applicable to a link is reinitialized upon restart (return to <ready> state). The statistics for instance will not be correct on a restarted link. The implication of this is that a link should be "purged" and a new "param" command issued rather than a "ready", if error-free operation of the Transponder is desired. (This missing "ready" feature is not necessary for the functioning of Transponder. It would take less than a major effort to repair this lack of re-initialization.)

"Start" requests that all links in the <ready> state go into the <active-wait> state. This will cause an acknowledgement to appear on the user's teleprinter. <Active-wait> indicates an open is being attempted on the network connection. [10] After a short period, [11] each of the links should transfer from the <active-wait> state to the <active> state, indicating that the connection was opened.

```
<ready>        --"start"--> <active-wait>*
<active-wait>  --network open-->     <active>*
```

"Status" causes the printing of the status table including the state of every link not in the <unused> state, the link number, and the time and date on which the state was acknowledged.

"Stop" is issued with a link number corresponding to a link in the <active> or <active-wait> states. The respective link is sent into the <stop-wait> state and is stopped (sent to the <closed> state) at the end of the current network message. [12] The <closed> acknowledgement indicates that the link's connection has been closed. The link may be restarted with a "ready" command.

```
<active-wait> --"stop" that succeeds--> <stop-wait>*
<active-wait> --"stop" that fails-->     <unused>*

<active> --"stop" that succeeds--> <stop-wait>*
<active> --"stop" that fails-->     <unused>*

<stop-wait> --internal processing delay--> <stop>
```

----

[10] Network Initial Connection Protocal begin initiated by the Unix NCP daemon. See Bolt-Berinek-Neuman document number 1822.

[11] Assuming a responder is available on the other "side" of the network.

[12] The "stop" command makes use of the Unix signal mechanism, which is a rather heavy handed device for this purpose, in the sense that it will abort most system calls. In particular it will abort most read or write system calls. Thus it cannot be guarranteed that the "stop" command will always allow the current network message to complete its travels.

"Purge" is followed by a single link number. The link is 'liquidated' (sent to the <unused> state via the <purge-wait> state) and the link number may be reused. The link will no longer appear in the table printed out by the "state" command. A purge acknowledgement should be seen by the user.

```
<closed>  --"purge"-->  <purge-wait>
<ready>   --"purge"-->  <purge-wait>

<purge-wait>  --internal processing delay-->  <unused>*
```

"Kill" is followed by a link number. It acts somewhat like a "stop" followed by a "purge" with the indicated link. However, unlike the "stop", the "kill" abruptly terminates network transmission and reception by the link. The link is liquidated. The liquidation of the link is not acknowledged--the link is immediately put in the <unused> state, effectively deleting it from the state table.

```
<purged>       --"kill"-->  <unused>*
<ready>        --"kill"-->  <unused>*
<active>       --"kill"-->  <unused>*
<closed>       --"kill"-->  <unused>*
<ready-wait>   --"kill"-->  <unused>*
<active-wait>  --"kill"-->  <unused>*
<purge-wait>   --"kill"-->  <unused>*
<stop-wait>    --"kill"-->  <unused>*
<unused>       --"kill"-->  <unused>*
<ERROR>        --"kill"-->  <unused>*
```

The "indirect" command, which may be abbreviated "i", is followed by a Unix file pathname. The contents of this file are interpreted as Transponder commands and immediately executed. "Indirect" and "i" commands may not appear in the file. The number of possible links is reduced by one 'inside' of an indirect file. [13] There is no printed acknowledgement associated with the "indirect" command itself.

The "param" or "p" command creates a link with the given attributes. It is one of the first commands that a user will probably issue. Its form is virtually identical to the Elf Transponder "param" command, right down to the unwieldly syntax:

```
param host, d1, d2, l1, l2, n, r
```

where host, d1, d2, l1, l2, n, r are unsigned octal numbers respectively representing the host address, message data pattern, an increment to the message data pattern, length of the initial message, length increment, number of messages, and repetition rate per message. The next section (The Format of the Test Data) describes the meaning of these parameters.

----

[13] See the Transponder Implementation Manual section on Transponder limitations for further information.

```
<unused> --"param"--> <ready-wait>
<ready-wait> --internal processing delay--> <ready>*
```

The numbers are separated by commas and should be range between 0 to 0177777 octal, with the exception of the host number, which must be in the range of 0 to 0377 octal. [14]

The Transponder will respond with a time and a link number (which it assigns). The link is put in the <ready-wait> state and eventually (after some internal processing) the user can expect an acknowledgement that the link is in the <ready> state.

2.5 The Format of the Test Data

The data is formated into messages, each message intended to be exactly one network message long. [15] The message is composed of a header and patterned data. The header is four bytes long. It begins with a byte whose value is 126 octal. This byte is known as the marker. The marker is supposed to be an unlikely data pattern, allowing some error recovery. Following the marker is a byte of sequencing information. This sequence byte increments from 0 to 377 octal and then back to 0 again. Again this allows for some error recovery. Immediately following the sequence byte is the high order (most significant) byte of a two byte number indicating the length (in bytes) of the patterned data. The low order byte of the length number immediately follows the high order byte of the length number. Finally, the next byte starts the patterned data.

The patterned data can be considered to consist of two byte words. There is no attempt to align these words on even or odd boundaries (of the start of the message). This generality has caused problems in the Transponder--see the Transponder Implementation Manual, the section on Transponder Limitations. The last 'word' of the patterned data may be truncated to its high order byte if the message size for this message is odd. The pattern will be picked up in the next message (presuming there is a next message) exactly where it is left off. This will mean that the next message

----

[14] Unix currently does not support the new imp-host header which allows 16 bit host numbers.

[15] The term network message is technical in nature--it refers to the 'block size' of the network connection. The size of a network message may, depending upon the host(s) involved, be fixed or variable, and may vary from 1 bit to the maximum of the number of bits permitted by Unix and the number of bits permitted by the network. (It would generally be a mistake for a host to imply data structure information from the network message size.) Unix always sends messages in multiples of 8 bits up to 8000 bits. Unix makes no attempt to use a constant network message size. Nor does Unix quarrantee that the user has control over the message size. The user may (using the Unix write system call) write a certain size message into system buffers, but depending upon various events, the buffer may be written in pieces or it may be combined with other buffered data (within the constraints of sequentiality). In probability, however, the size of the network message will correspond to the size of the buffer specified in the write system call. The Transponder takes the size of its 'write' buffer to be the size of the network message.

will have a different boundary than the current one (even if the current one is odd, and odd if the current one is even).

2.6 Files used by the Transponder

Various statistics are kept on link performance. For further information, see the parallel Implementation Manual section.

A copy of the user's commands along with some pertinent responses, logged with the time and date, are kept the file transponderlog in the working directory (in which the the Transponder was started).

The NCP can also keep a log file (named, by convention, /usr/lpd/ncplog) if so instructed [16] This file is, by convention at Illinois, /usr/lpd/ncplog.

----

[16] See the ncplog Manual page of the Unix Programmers Manual (Section VI). for information on how to start and interpret this log file.

2.7 Additional Commands

The exclamation point command can be used to perform some 'Transponder' commands. For instance, to print the receive statistics for link one, the user may type

```
!cat r-slave1
```

after the link has entered the <closed> state. Or to print all statistics accumulated in the current working directory, use:

```
!cat [rt]-slave*
```

The user can make use of the exclamation point command 'inside' of the Transponder to start the Responder: "!/etc/responder>/usr/lpd/responderlog&" is identical to "listen".

For more information see the Unix Manual description of the shell.

2.8 The Function of the Responder

The Responder program is the echo daemon that listens on network socket SOCKNUM (viz. seven) for the opening of a duplex connection. Once the connection is established, the Responder retransmits every message that it receives (and no others). The Responder keeps statistics about the number of messages read and written as well as their rates. The Responder writes the statistics to the Unix standard output upon the closure of the connection.

3 Transponder Maintenance Manual

3.1 Installation

The sources for the Unix Tranponder can be found in directory ncprogs/xponder. In this directory you will find the files master.c, slave.c, rbuf.c, transpond.h, slave.h, socknum.h, and responder.c . These sources have been compiled on a number of different C compilers, some of which accept different input syntax.

These sources are compiled with the following commands (using the Illinois C compiler and auxiliary library /lib/libj.a):

```
cc master.c -fOsnlj
cc slave.c rbuf.c -fOslj
cc responder.c -fOsnlj; mv responder /etc
```

If the Illinois C compiler is not available, but the standard version six C compiler is, the following command may be used (still assuming /lib/libj.a):

```
cc -f -O -s -n master.c -lj; mv a.out master
cc -f -O -s    slave.c rbuf.c -lj; mv a.out slave
cc -f -O -s -n responder.c -lj; mv a.out /etc/responder
```

Or if the Version seven Unix compiler is used:

```
cc -f -O -s -n master.c -o master
cc -f -O -s    slave.c rbuf.c -o slave
cc -f -O -s -n responder.c; mv a.out /etc/responder
```

If the Illinois compiler is not used, [17] the resultant executable files may not work due to bugs in some versions of the C compiler and problems in the printf routine. In particular the long decimal print ("%D") will not work. [18]

Also note that if the Transponder is to be run on a system with a floating point processor, the "-f" switch need not be specified. Also note that the "-n" switch (specifying reentrancy) is not specified with the compile of the slave. This is because on at least some non-floating point machines, the "-n" option causes premature termination of the slave. This is very unfortunate, since by all logic it is the slave that should be reentrant and shared. The cause of this bug may be the compiler used or a bug in the floating point interpretation. However, at this time there is no evidence one way or another. [19]

----

[17] or the Unix version seven compiler and associated library.

[18] Libj.a contains a printf which has "%D". This is the main reason for using libj.a

[19] It is worth while to experiment with your compiler to see if the "-n" switch can indeed be used with the Transponder, as it will increase the Transponder's performance.

The file slave.h contains macro defines common to the files slave.c and rbuf.c that constitute the Slave sources. The file transpond.h contains macro and structure defines used by both the Master and the Slave sources.

The compiles will result in the production of three executable files: master, slave, and responder. Master and Slave should be in the directory from which the Transponder test will be conducted. Master may be in this directory also, or it may be put in a system binary directory such as /usr/bin.

The Responder and the Slave must be recompiled if it is necessary to change the socket on which they listen. This is considered acceptable, since this operation requires only a few minutes and since it is infrequently done (if ever). There is only one define--SOCKNUM, to be changed. SOCKNUM resides in socknum.h .

The source listings included later in this document are indexed. The indices are of the form NNN-MMMM where NNN is a number unique to each source file and MMMM is the number of the line within the source file. The index number for the Transponder sources are:

170	responder.c
180	transponder.h
181	master.c
190	slave.h
191	slave.c
192	rbuf.c
193	socknum.h

4 Transponder Implementation Manual

4.1 Transponder Process Structure

The Transponder functions as several processes. Conceptually it can be decomposed into three sets of processes--Master, Slave, and Responder. [20] There is what appears to be a monolithic Master process and many Slaves, one Slave for each network connection to be exercised. The Master interprets commands issued by the user. There are commands for creating, deleting (liquidating), and controlling Slaves. The Master may, depending on the command, send controlling messages to zero or more Slaves. The affected Slaves obey the Master by changing states. Each of the Slaves then sends an acknowledgement back to the Master, indicating the state they have entered.

4.2 Master-Slave Status Acknowledgements

The Master keeps track of the status of each of the Slaves, along with the time at which the status last changed. There are eight possible status values, five of which correspond to states of the Slave (as received via acknowledgements received from the reply pipe described below) and the other three of which are intermediate status values, indicating that a second state change is imminent. The status information is kept in a table known, quite naturally, as the status table. Slave processes are made synonymous with links as used in this document. Controlling messages from the Master are acknowledged not only for timing purposes but also for verification purposes. The Master will print an interpretation of the acknowledgement on the user's teleprinter as it is received. The printed acknowledgements are preceded by a left angle bracket and followed by a right angle bracket. They contain the link number of the Slave process and the type of acknowledgement. The type can be one of purged, ready, active, closed, ready-wait, active-wait, purge-wait, stop-wait, or ERROR. These will appear on the user's teleprinter in the format:

```
<L ack>
```

where L is the link number in decimal, and ack is the particular status type. In this document the status type will be surrounded by angle brackets. This is to distinguish them from commands, which will be surrounded with double quotes. In actuality <ready-wait>, <purge-wait>, and <stop-wait> will never be sent as acknowledgements, but will appear in the status table. The state <unused> will not be printed either. Also, if a Slave is in the <unused> state then the Slave's data will not even be printed by the "status" command. If an expected acknowledgement is not forthcoming, the Transponder is not functioning correctly. (Check, for instance, to see that all the requisite programs are present.)

----

[20] When a process is refered to in these Manuals, it will be capitalized. This is to help distinguish it from any binary file with the same name (which will not be capitalized).

4.3 Responder Internals (Process Structure)

The Responder program is the echo daemon part of the Transponder. It listens on network socket SOCKNUM (viz. seven) [21] for the opening of a duplex connection. Once the connection is established, the Responder retransmits every message that it receives (and no others). The Responder keeps statistics about the number of messages read and written as well as their time intervals. The Responder writes the statistics to the Unix standard output upon the closure of the connection.

The Responder is very similar in design to server-telnet, only somewhat simpler. The main procedure (main‖responder.c) [22] is responsible for the initial network open. This open will succeed when someone connects to the listen socket. When this happens, a fork is done and the parent loops back to the open, allowing multiple connections to the socket. The child process calls the procedure responder‖responder.c . At this point another fork is done and the parent dies, leaving the child an orphan. [23] This prevents the death of various Responder processes from clogging the process table. Procedure copier‖responder.c is then called from responder‖responder.c . Copier‖responder.c does all the work for the Responder. It is a loop with counting and time-keeping code surrounding a read and a write statement. Copier‖responder.c executes the read/write cycle until the read returns a -1, the standard Unix error return. At this point copier‖responder.c assumes the connection is closed; it writes out the available statistics and exits.

A signal four sent to the Responder will cause it to 'dump' itself (write a file named core with its image, in the current working directory), so that it can be debugged. A signal five will cause the Responder to die without a whimper.

4.4 Master Internals (Process Structure)

The Master program interfaces the Transponder with the user's terminal. It decodes user commands and (when appropriate) passes them on to other processes known as Slaves which do the actual work of data formation, transmission, and verification.

----

[21] SOCKNUM is a compile time parameter. Socket seven has been chosen as a defacto standard for 'echo' sockets by various ARPANET Tenex sites. However, the Elf Transponder chose socket number thirteen for its Responder, so whenever the the Unix and Elf Transponders are interfaced, one or the other will have to be changed. SOCKNUM is defined in socknum.h .

[22] Whenever a program name (i.e. procedure name or global structure, or etc) is referenced it will be (in at least its first appearance, followed by two vertical bars ("‖") and by the source filename in which it resides.

[23] Until a defunct process is claimed by its parents, it stays in the process table. All orphans are claimed by process one, /etc/init .

The Master is a very peculiar Unix program, because of the lack of asynchrony in Unix input/output. The Master must wait on the inclusive-or of two events: input from the user keyboard and input from subprocesses. There is only one way to implement this in standard Unix--that is with a pipe with many writers and one reader. Let us call the pipe a reply pipe.

These writers are 1) a process that we will call the Echo Master [24] and 2) zero to fifteen (decimal) Slave processes. [25] The Echo Master sits in a tight loop reading from the user's keyboard and writing on the reply pipe in procedure initmaster.c . The Slaves put messages in the reply pipe only in response to some stimulus from the Master.

The reply pipe messages must be kept short, so as not to be garbled, and there must be some way of distinguishing between the various writers. This was accomplished by the use of one byte messages with the high order bit distinguishing between bytes originating from the user's keyboard and bytes placed on the reply pipe by Slave processes. Advantage was taken of the fact that Unix forces user terminals to appear as seven-bit ASCII devices. This means that input from the user's keyboard can be written on the reply pipe exactly as read from the terminal and once written, can be distinguished by the high order bit of the byte--it must be zero for it to be input from the user (via the Echo Master), else it is input from a Slave. That is, user input to the Master is of the form:

0X XXX XXX

where the above represent bit positions with 0 off, 1 on, and X's encode the ASCII character data.

Characters from the user's keyboard are read by a process that shares the Master reentrant image--it will be called the Echo Master. The Echo Master puts these characters on the pipe as it receives them. [26]

----

[24] The Echo Master is really a child of the Master, sharing the same reentrant code.

[25] The number of Slaves is really limited to somewhat less than fifteen by certain factors mentioned later in the section on limitations.

[26] The Echo Master should be the process to interpret the "!" command (see the Transponder User's Manual) even though currently it interprets no commands. The reason for this is that the Echo Master could pass the user's keyboard to the sub-shell as an input file rather than as a pipe, as it is done currently. Then all commands would work as expected. The current implementation causes certain programs (those that expect to be able to set raw mode, for instance) to behave unexpectedly. Also no end of file, interrupt, or signal can be signaled from a pipe, thus the user must be careful not to execute programs that use those 'out-of-band' signals for information. For instance, cdb should not be run from the Transponder as the only way to exit from cdb is with an end of file! If the user gets him/herself into such an unfortunate dead-end situation, he/she can extricate him/herself by using the quit signal. This will kill the Echo and Command Master. It will generate a core image, but it will work.

The Command Master acts in response to the data it reads from the reply pipe. It accumulates input from the user until a full line is accepted. (This is done in procedure commands$master.c .) If it receives data from a Slave process, it updates the array CONTROL$transpond.h . This array thus holds the current state of each Slave as the Master perceives it, and is the 'status table' mentioned in the Transponder User Manual.

A Slave process is created every time the user types a "param" command. This Slave is given the read end of a pipe whose write end is connected to the Command Master. Let us call this pipe a command pipe to distinguish it from the aforementioned reply pipe. The Command Master gives the read end of a different command pipe to each Slave created. Note that the write end of the reply pipe is passed also. [27] The CONTROL array keeps track of which Slave corresponds to which command pipe file descriptor. The Master controls an individual Slave by writing a single byte down the respective command pipe.

Data from Slave processes, as they appears on the reply pipe, are of the form:

1Y YYY ZZZ

where the bits represented by the Y's encode the particular Slave subprocess (0 to 17 octal) and those represented by the Z's encode the data.

Three commands are defined to fill the Z's in the above format. They are (as found in transpond.h):

READY_CM (viz. 122 octal, that is, the character 'R') indicates that the Slave is to go to the <ready> state. It is sent on response to a "ready" command. This is supposed to reinitialize a Slave in the <closed> state. (See discussion of the "ready" command in the Transponder User's Manual.) It however does not work as anticipated. (See discussion of the "ready" command in the Transponder User Manual.)

ACTIVE_CM (viz. 101 octal, that is, the character 'A') indicates that the Slave is to go to the <active-wait> state. It is sent to all <ready> Slaves in response to a "start" command. This state indicates that a connection is being attempted.

PURGE_CM (viz. 120 octal, that is, the character 'P') indicates that the Slave is to commit suicide. It is sent in response to a "purge" command.

The procedure structure of master.c is very simple. It is composed of a command dispatch routine, a set of command routines, and a set of auxiliary routines. The command dispatch routine is named commands$master.c . It dispatches via a table with the name comm$master.c . Comm$master.c is structured as a (string, procedure name) pair. The procedure commands$master.c has an additional function-- it will dispatch to the Slave reply routines if it sees a reply from a Slave (a byte with the high order bit set), since it contains the loop waiting (polling) for reply pipe input. [28]

----

[27] Of course, the fact that Unix hands the reply pipe to every child process makes the pipe command-reply scheme work.

4.5 Slave Internals (Process Structure)

The Slave source files (slave.c and rbuf.c) is (in the program form) the process(es) that are under the command of the Master. Many Slaves processes may be active while the Transponder is running. There will be a pair of Slave processes per duplex network connection (link). One process of this pair will transmit on the send side of the duplex connection. This process will be referred to as t-slave. The other process of the pair will receive data on the receive side of the connection. This process will be referred to as an r-slave. [29] Each t-slave has the responsibility of formating the data which will be transmitted across the net to its respective r-slave. The r-slave has the responsibility for receiving and checking the data.

Both Slaves report statistics about their activity, in a similar manner to the Responder, except of course that only the write behavior is reported by the t-slave and only the read behavior by the r-slave. The r-slave also reports any incorrect data that it receives.

The Master actually only creates one Slave per link (per "param" command). The Slave thus created replicates itself to form a r-slave (the child) and a t-slave (the parent) upon receipt of an ACTIVE_CM command. There is no communication between the r-slave and the t-slave other than over the network connection that they share. [30]

The transmitted data can be checked without any other communication between the t-slave and the r-slave because both Slaves have knowledge of the param statement that created them and the data pattern is a function solely of the param statement.

----

[28] There is one other procedure that dispatches to Slave reply routines. It is verify_master.c .

[29] In fact these processes will alter their own argument list so that the Unix command ps will print more meaningful data. Their zeroth argument will be "r-slave" or "t-slave" depending upon their function and will be suffixed by their link number. Another way of putting this is that the statistics file that the process opens will have the same name as their zeroth argument as seen by ps. Ps will also display a set of arguments which represent (in this order) the process id of the process, the value of the file descriptor that constitutes the command pipe, and the list of param arguments.

[30] Thus, of course, the Master could have exec'ed them individually...

4.6 Files used by Transponder

The Transponder will create several files. These include certain files that report on the statistics gathered by the r-slaves and t-slaves.

There is one file for each Slave. These are created in the current working directory (at the start of Transponder execution) under the names r-slave1, t-slave1, r-slave2, t-slave2, and so on. The files starting with 'r' are statistics on the reception of data, gathered by the r-slaves. The files starting with 't' are statistics on the transmission of data, gathered by the t-slaves. The suffixing numbers (they are decimal) denote the link number of the corresponding Slave.

Each file is created (truncated) with the creation of the Slave. It is updated for the most part (excluding errors and a one line header) at the Slave's entry into the <closed> state. The t-slaves' files grow with each "ready", "start" sequence (when the link is made <active>). The r-slaves' files are created every time the link is made <active>.

Besides r-slave and t-slave files, the Transponder appends a log style script to a file with the name transponderlog.

The Responder portion of the Tranponder will create a file with the name responderlog. This file will be in directory /usr/lpd. It is worth noting that the NCP daemon has a log file with the name /usr/lpd/ncplog.

5 Transponder Limitations

5.1 Number of Slaves (Links)

The Transponder has a designed in limitation of 15 (decimal) Slaves. This is not the only limitation on the number of Slaves. The most severe limitation is the number of file descriptors a process can have. This limits the number of Slaves to six, if they are created with param statements in an indirect file, or seven otherwise.

In general more than one Transponder (Master) will have to be run if more than the upper limit of connections are needed.

The Transponder assumes no special clock device. It thus has only a one second timing resolution, as provided under standard Unix. A source module by the name of clock.c is included in the ncprogs/xponder directory. It contains an experimental attempt to access the line frequency clock. It was never integrated into the Transponder because of contract time limitations. As a consequence of this, the timing statistics are to be taken as significant only over the long run (that is the user should send a large number of messages in order to get timing information).

As another consequence of the lack of time resolution, the rate argument on the "param" command is valid in the long run. Also, a non-zero rate argument will probably result in a 'bursty' demand curve.

5.2 The Use of Pipes for Inter-process Communication

The pipe structure of the Transponder is rather awkward, causing delays, unnecessary disk activity, and tying up system buffers. A better solution would be to look beyond standard Unix to an Inter-process Communication facility.

Because the Slave cannot be reading from both the command pipe and doing I-O on the network file, the Slaves are not able to perform certain functions. The Slaves essentially ignore the command pipe during the entire time that they are doing network I-O. Also without elaborating the data header format, the Slaves are not able to keep 'round trip' timing statistics--they are limited to inter-message intervals only. The lack of a good Inter-process Communication facility also means that the Slaves are not able to dump their statistics upon demand of the user. [31]

5.3 Data Checking Overhead

----

[31] Round trip statistics could be collected by putting a timestamp in an (expanded) header field.

Another limitation is the processing time the r-slave requires to check the received data. The format of the data (taken from the Elf design for compatibility reasons) was a poor choice for the PDP-11.

The data pattern designed for the Elf Transponder is a word oriented one, (that is, it uses an even numbers of bytes) whereas the Transponder does not disallow the use of odd message sizes. Further, when a Unix user process (such as the Transponder r-slave) reads from the network it requests a number of bytes and it is handed the minimum of its request and the available data from the network. There is no attempt to split data coming in from the net on network message boundaries, so if the user process requests enough data, it may receive more than one network message. Since the Transponder r-slave specifies a large byte request on its network reads, it often gets multiple messages in a single read. [32] Now if the Transponder test involves odd message sizes, the word data may or may not then appear on word boundaries. The data may actually appear on 'in phase' (on word boundaries) and 'out of phase' (on odd byte boundaries) in the same read buffer!

It requires either a rather complex algorithm for the recognition of correct data, or multiple procedure with several calls per byte of received data. The use of multiple procedures was used in the Unix Transponder, as this made for the cleanest, easiest to maintain code.

Because of this problem in data checking, the Transponder seems to have a limit of about twenty kilobaud when run on an PDP-11/70. This is fortunately approaching the upper limit of the fifty kilobaud ARPANET. This limit is too low for other, faster networks.

There are several solutions to this problem. One of the easiest is a new C compiler optimization (being developed at Illinois) which would speed up some procedure calls. Another solution is to make data checking optional. This, needless to say, is not a very satisfactory solution. Another would be to rewrite the data checking code, at the expense of clarity. One of the most practical solutions is to recognize that the Elf data patterns are simply inefficient and to design more satisfactory pattern, one that could, for instance, make more use of PDP-11 word instructions and less use of byte instructions.

----

[32] It should be noted that the Unix Transponder r-slave allocates buffer space for the incoming data ont eh basis of the message size and message increment as specified in the "param" command. It attempts to allocate for two messages of the maximum size that it expects. This scheme is in contrast to the Elf scheme which requires reassembly and rebooting of the whole system in order to change the message size.

6 Sample Transponder Session

```
        Begin Typescript of /dev/ptyA at Mon Aug  1 21:19:57 1977


Center for Advanced Computation
     Network Unix System
Login: balocca
Password:
Last login Mon Aug  1 21:14:50 1977
% master
COINS Transponder for Unix      Mon Aug  1 21:20:39 1977

$ : This is an example of the use of the Transponder
$ help
Commands are:
help          quit          status        kill #
date          time          purge #       !<Unix>
param #,#,#,#,# start          stop #         ready #
p #,#,#,#,#     indirect file   i file        pause #

param arguments are:
host number, pattern, pattern increment, message length,
length increment, number of messages, repetition rate
$ status
$ param 114,0,1,0,1,100,0
<1 ready>
Assigned link is 1
$ status
 1 ready Mon Aug  1 21:21:55 1977
$ start
<1 active-wait>
$ Aug  1 21:22:04 TR(47): Begin transponder r-slave

<1 active>
$
<1 closed>
$ status
 1 closed Mon Aug  1 21:22:11 1977
```

```
$ !cat r-slave1
Aug  1 21:22:04 TR(47): r-slave1: 76 0 1 4 1 64 0
Aug  1 21:22:10 TR(47):          24 Messages read/written, 0 errors
Aug  1 21:22:10 TR(47): Inter-message intervals in seconds:
Aug  1 21:22:10 TR(47): Minimum:          0
Aug  1 21:22:10 TR(47): Maximum:          1
Aug  1 21:22:10 TR(47): Average:        0.250
Aug  1 21:22:10 TR(47): Total:          6
Aug  1 21:22:10 TR(47):
Aug  1 21:22:10 TR(47): Message lengths in bytes:
Aug  1 21:22:10 TR(47): Minimum:          4
Aug  1 21:22:10 TR(47): Maximum:        952
Aug  1 21:22:11 TR(47): Average:         94
Aug  1 21:22:11 TR(47): Total:         2272
Aug  1 21:22:11 TR(47):
Aug  1 21:22:11 TR(47):  4.00 Messages/second,  3029 Baud
Aug  1 21:22:11 TR(47):        Connected     6 seconds
Aug  1 21:22:11 TR(47): End transponder r-slave
$ ready 1
<1 ready>
$ start
<1 active-wait>
$
<1 active>
$
<1 closed>
$ pause 10
$ i t
<2 ready>
Assigned link is 2
EOF on command file
$ start
 1 closed
<2 active-wait>
$ Aug  1 21:24:02 TR(99): Begin transponder r-slave

<2 active>
$ status
 1 closed Mon Aug  1 21:22:46 1977
 2 active Mon Aug  1 21:24:03 1977
$ time
21:25:06
$ date
Mon  Aug  1, 1977
$ status
 2 active Mon Aug  1 21:24:03 1977
$ kill 2
$ status
 1 closed Mon Aug  1 21:22:46 1977
$ purge 1
```

```
l purged>
tatus
it
e



d Typescript of /dev/ptyA at Mon Aug  1 21:26:27 1977
```