To: Distribution

From: Robert S. Coren and Michael J. Grady

Subject: SOFTWARE FOR INTER-SYSTEM FILE TRANSFER

Date: 03/01/74

This document describes a design for software to enable files to be transmitted between the MIT and Phoenix Multics systems, using the new I/O Daemon mechanism. Comments should be addressed to Robert S. Coren at CISL, or by mail to Coren.Multics.

This document is divided into the following sections:

I. OVERVIEW
II. COMMAND INTERFACE
III. NORMAL OPERATION OF FILE TRANSFER DRIVER
IV. FILE TRANSFER DIM -- METHOD OF DATA TRANSFER
V. INITIALIZATION AND TERMINATION OF DRIVER PROCESS
VI. BRIEF SUMMARY OF PROGRAMMING REQUIREMENTS
VII. SUMMARY OF DATA MESSAGES AND ORDER CALLS

I. OVERVIEW

To facilitate the speedy transfer of segments between two Multics systems, we propose to make use of the new I/O daemon mechanism to implement a "transfer_file" (tf) command that queues requests to have files transmitted along a synchronous telephone line. It is not expected that this facility will be the primary method for the transfer of files from one system to the other, but rather that it will be available for the transferral of files which are needed more quickly than is possible through the normal use of the carry-tape facility.

In the initial implementation (at least), only segments will be transferrable by this facility (not directories and not multi-segment files). ACLs and additional names will not be transferred. The capability to transfer multi-segment files may be added later, but its use is not recommended because of the large amount of time required for the transmission of large files.

II. COMMAND INTERFACE

The "transfer_file" command will have the effect of a "dprint" command with a special "-device_class" argument; that is, it will place a request in a daemon queue that will be processed by a particular type of I/O daemon driver. The command usage will be something like the following:

transfer_file -control_arg1- path1 -control_arg2- -path2- ...

path*i* are the pathnames of segments to be transferred.

control_arg*i* may be either of the following:

-not_ascii, -na The segment may contain non-ascii data, e.g. it might be an object segment. If this control argument is supplied, the data will be transmitted in 6-bit "bytes" (see discussion of data transfer in section IV); otherwise it will be transmitted as 4 ascii characters per word. This control argument should *not* be used for ascii data, since it decreases transmission speed significantly; however, if this control argument is omitted, non-ascii characters will not be transmitted.

-queue *n*, -q *n* The request is to be placed in priority queue *n*, where $1 \leq n \leq 3$. The default is 3.

The transfer_file command will probably handle its control arguments in a manner similar to dprint, i.e. control arguments will apply to subsequently-named segments.

III. NORMAL OPERATION OF FILE TRANSFER DRIVER

A special kind of I/O daemon "device driver" process will run on each system; the two processes will be essentially identical, and each will be capable of either sending or receiving segments. If the queues at one end have requests in them and the queues at the other end are empty, the process with the full queues sends and the other one receives. If the queues at both ends are non-empty, the two processes take turns sending (the mechanism for this is explained further on). To determine which one sends first initially, one of them (through selection of the appropriate process-overseer entry point) is designated as the "master" process and the other as the "slave" process; if both driver processes come up with non-empty queues, the "master" process gets to send first.

If the queues at both ends are empty, the processes take turns checking their queues; the one that first finds a request gets to send first.

A file transfer driver during normal operation can be thought of as alternating between two modes, "send" and "receive". A brief description of the proposed operation of these two modes follows.

"Send" mode

The driver enters "send" mode when it receives the standard "request ready" wakeup from the I/O Coordinator (see MTB-004 for details). (As far as the coordinator is concerned, a file transfer driver is just another output driver.) Upon receipt of this wakeup, the driver prepares a "header" for the segment to be transferred; this header contains the name of the segment, its length in characters, the person-project-id of the requestor, and an indication of whether or not the segment is all ascii. An order call causes this header to be transmitted to the receiving process.

The actual segment, followed by an "end-of-segment" message, is transmitted by the File Transfer DIM in response to a "write" call; the sending process then performs whatever accounting is desired, and sends its coordinator a wakeup informing it that the request has been completed. The bit used by the driver to tell the coordinator whether it is ready to perform another output request is set to "0"b before this wakeup, because the driver will now switch to "receive" mode.

"Receive" mode

To enter "receive" mode, the driver issues an order call which enables it to receive a segment header from a sending process at the other end of the line. (If instead of a header it receives a special message indicating "no request to send", it sets its "ready for request" bit on and enters "send" mode if its coordinator provides it with a request.) It then receives the actual data by means of a "read" call.

The segment thus transmitted is placed in a subdirectory in a designated "file transfer" directory (which will initially be >udd>Multics>Transfer). Each user of the file transfer facility is given a private subdirectory in which his/her segments are placed; this avoids conflicts if two users happen to transmit segments with the same entry name. Once the segment has actually been transferred, the requestor receives mail on the receiving system informing him of the fact.

Suppose, for example, that Smith.Multics, logged in on the MIT system, issued the following command:

```
transfer_file >udd>Multics>Smith>foo.pl1
```

The data would be put in a segment on the PCO system with the pathname

```
>udd>Multics>Transfer>Smith>foo.pl1
```

and Smith.Multics on the Phoenix system would receive the following mail:

```
From Transfer.Multics:
foo.pl1 has arrived in >udd>Multics>Transfer>Smith
```

Smith.Multics would have "sm" access to his subdirectory in the file transfer directory, so he could copy the segment into his own hierarchy and delete it from the file transfer subdirectory. He would not, however, have "a" access to this directory; therefore, he cannot plant links to other parts of the hierarchy in the file transfer subdirectory.

While a segment is being received, the file transfer driver sets access on it to "null" for all other users; this prevents the requestor (or anyone else) from accidentally picking up the segment in an inconsistent state.

Note that if Smith.Multics transfers foo.pl1 twice, or transfers two different segments with the same entry name, the earlier one will be overwritten by the later one at the receiving end.

Once it has received a complete segment, the file transfer driver wakes up its coordinator to advise it that it is ready to send another segment; if the coordinator provides it with one, it enters "send" mode and proceeds as described above; otherwise it sends the special "nothing to send" message mentioned earlier, and remains in "receive" mode.

The alternation mechanism described here leaves us with the problem of what happens when the request queues at both ends are empty, and in particular how transmission can be recommenced once a request appears in one queue or the other. The solution requires that the two drivers take turns checking their queues, so as to avoid having both of them in "send" mode at the same time; however, we wish to avoid the excessive waste of resources that would result if these checks were made as often as possible.

Suppose that MIT has just completed sending a request to PCO, and that the queues at both ends are now empty. the following steps would be taken:

1) The PCO driver interrogates its queues (through the coordinator), finds them empty, and sends the "nothing to send" message described above. It then prepares to receive a request header in response.

2) The MIT driver, on receiving the "nothing to send" message, interrogates its queues; since they are empty, it sends back a "nothing to send" message.

3) The PCO driver now notices that nothing has been transmitted either way since it last checked its queues; therefore it goes to sleep for a short interval (say 30 seconds) before checking its queues again. If they are still empty, it then sends another "nothing to send" message.

4) After receiving this message, the MIT driver likewise goes to sleep briefly before checking its queues again: assuming they are still empty, it replies with yet another "nothing to send" message.

Steps 3) and 4) are repeated until one of the drivers has something to send (or terminates the session, see Section V). It will then enter "send" mode and transmission will proceed as before. The receiving driver will turn off the flag that tells it to go to sleep before the next time it checks its queues.

The salient feature of this mechanism is that neither driver ever checks its queues unless the other is in "receive" mode; thus a driver that finds a segment ready for transmission can immediately begin sending it without fear of a conflict.

Operator Input to a File Transfer Driver

The normal operation of a file transfer driver never requires operator input, and the driver will not check for input through the "read_status" order call after each request, as a regular device driver does. It will accept input after a QUIT, after which all commands acceptable by a device driver ("start", "kill", "restart", "logout", etc.) can be entered, but the utility of actually sending a file transfer driver a QUIT and a command seems rather limited. The primary form of outside "control" over the file transfer driver process will be the "switch_driver" command described in Section V of this document.

IV. FILE TRANSFER DIM -- METHOD OF DATA TRANSFER

This section will be devoted to the discussion of the file transfer DIM(ftd_) and the messages it will transmit and receive.

Message Formats

The design of the messages to be used in the file transfer process was simplified as much as possible in order to avoid further changes to the ring 0 tty_dim and the 355 software. Thus a good part of this design was taken from the message format used by the GRTS Remote Computer Interface, which was discussed in MSB-110. The GRTS design allows the transmission of ASCII data over synchronous communications channnels, with termination on ETX followed by a block check character. Hopefully, with this design, no changes will be required to either the ring 0 tty_dim or the 355 software.

The message format itself consists of a header, containing control information, followed by up to 500 characters of text, terminated by an ETX character and a block check character. These messages will be transmitted over a synchronous line, 7 bits per character with odd parity generated and checked by the 355. This parity check plus the additional check by the block check character will allow detection of all 3(or less) bit errors and most other multiple bit errors. Error correction will not be possible with this scheme since the 355 only notifies us of the parity error, but not which character. However, detection is the most important feature since the message protocol allows retransmission of messages received in error through the use of ACK/NAK responses. The projected error rate of the communications channel is 1 in 10,000 bits and with approximately 500 character messages, we have an error rate of about 1 in 15 messages. This appears acceptable and we should be able to recover with one retransmission.

Messages transmitted will have the following format:

syn syn syn syn soh fc sc oc stx -text chars(0-500)- etx bcc

Where:

syn	is a synchronization character used to allow the receiving 355 to begin accepting characters.
soh	is a start of header character(001).
fc	is a format code character used to indicate the type of message, explained below.
sc	is a sequence code, either 101 or 102, and is

	used to prevent reprocessing of retransmitted data.
oc	is an operation code containing an acknowledgement field and a command field, explained below.
stx	is a start of text char(002).
text chars	are ASCII text characters which may be present in a message. The various types of text are explained below.
etx	is an end of text character(003).
bcc	is the block check character, used to check the message for errors, also explained below.

The format code is used to describe the various types of messages which can be transmitted. A format code of 101 indicates a service message. Service messages contain no text characters and are used primarily to send commands to the other system. A format code of 104 indicates a segment header message, and the text contains a packed header for use by the driver process. The third format code, 110, is the text message code and is used when sending the actual segment.

The operation code contains two fields, the acknowledgement and the command. The acknowledgement field is used to indicate whether or not the last message was received correctly. A non-zero acknowledgement means an error, and that the message should be retransmitted. The command field is used to request special actions or to indicate a particular state of operation. The format of the operation code is as follows:

field	meaning
unused(1 bit)	always on to make the code ASCII.
ack(3 bits)	the acknowledgement field.
cmd(3 bits)	the command field.

The no-operation command(nop) is used when no special action is required other than the processing of any data which may be present in the message. Commands are defined on a per format code basis, i.e. each format code has an associated set of command codes. These are as follows:

for service messages(fc = 101):

nop	"000"b
ready_to_attach	"001"b
attach_accepted	"010"b
end_of_seg	"100"b
queues_empty	"101"b
end_of_session	"110"b

for header messages(fc = 104):

nop	"000"b

for text messages(fc = 110):

nop	"000"b
ascii_data	"001"b
packed_data	"010"b

The service messages are used to indicate the state of the driver process and are normally sent as a result of an order call. The commands for text messages are used to indicate the type of text in the message.

The message text consists of the actual data to be transmitted, which may be in two forms.

1) The data may be all legal ASCII characters, which will be transmitted as is. Data compression will be used where possible and the data will be checked for two special characters, US(037) and ETX(003), which are used by the system to indicate compression and termination, respectively. If either of the characters are found in the text to be transmitted they will be ignored. The compression code will be used when 3 or more of the same characters are to be transmitted. When this occurs the following format will be used:

US char count

where US(037) indicates compression, char is the character being compressed, and count is the binary count in the range 0 to 63 (0 to 77 octal) with the high order bit, bit 7, always on (to make the character transmittable ASCII).

2) The data may be also be any 9 bit byte, which could not be transmitted as ASCII. In this case the data will be packed so that it may be transmitted by taking 6 bit bytes from the source and transmitting them as 7 bit characters with the 7th bit always on to allow correct transmission of special characters. The compression technique explained earlier will be applied to these packed characters also.

Thus any segment can be transmitted using this scheme, with decreased throughput for non-ASCII data.

The block check character is an additional check made on received data to insure proper transmission. It is the exclusive or of all the characters in the message after the SOH character, and will be generated and checked by the DIM.

DIM Design

The DIM will consist of one I/O system interface module and four main device control modules, similar in design to the g115_dcm_. The interface module will make calls on the device control modules and will be responsible for going blocked waiting for events to occur. The four main device control modules are: the attach/detach module, which will also handle the order calls; the read and write modules to do the actual device I/O; and an event-call-driven module to handle wakeups from the ring 0 tty_dim. This module will be responsible for acknowledging messages and sending wakeups to the I/O system interface module.

Attaching

The DIM will always be attached in read and write mode, and two other modes will also be permitted. The first is "notify" mode which tells the DIM to abort the attach after five minutes if unsuccessful. The other mode is either "master" or "slave" which indicates which process is to control the attach sequence of messages.

After initialization the attach module will send a "ready_to_attach" message. This will be repeated every 30 seconds if the mode is "master", and every 45 seconds if the mode is "slave". Note that if both processes transmit at the same time, both messages will be lost and thus the time difference is required to allow the eventual correct transmission by one or the other. Several different responses can be expected depending on the line connection and the mode of the transmitting process (i.e., the first process to successfully transmit a "ready_to_attach" message which is received without error). If the line is incorrectly set the process may receive an MDS2400 type NAK message. If this occurs the attach will be aborted with

an appropriate error code. If the transmitting process is the "slave" then the "master" will send another "ready_to_attach" message, the "slave" will then send the "attach_accepted" message and await the same from the "master". Otherwise if the transmitting process is the "master", the "slave" will send the "attach_accepted" message and the "master" will respond with the same. Thus the "slave" is always the first to send the "attach_accepted" message.

Sending

The technique of writing a segment to the other process will consist of an order call to send a header message, followed by a write call to write the entire contents of the segment.

The order call that writes the header will block until the receiving process sends the acknowledgement, indicating that the data may be transmitted.

The write call will break the data down into 400-500 character messages, convert as explained above and transmit, one message at a time, awaiting the acknowledgement before sending the next message. At the completion of transmission the "end_of_segment" service message will be sent.

If an error occurs during transmission of a message the response will be a NAK, and we will retransmit the last message.

If the acknowledgement message was received in error, we will retransmit the last message, this time with a NAK. This allows the other process to transmit its acknowledgement message again.

Receiving

When preparing to receive a segment the following calls will be made: an order call to receive the header, followed by a read call to read in the segment and place it in the file system.

The order call to receive the header will block and wait for one of the following messages: a header message, a queues_empty message, or an end_of_session message. When the message is received an acknowledgement is sent and the message type is returned to the caller.

If a segment is being sent a read call is made to the DIM and the messages are unpacked and placed in the specified segment. If no errors are found then an ACK is sent, and the DIM blocks and waits for the next message. Otherwise a NAK is sent and the DIM waits for the retransmitted message.

Upon receipt on the "end_of_seg" message the DIM returns the

number of elements read.

Detaching

No messages are sent as a result of the detach call, and the device is detached.

V. INITIALIZATION AND TERMINATION OF DRIVER PROCESS

Certain problems are posed by the actual physical setup at CISL. In particular, we have only one high-speed line to Phoenix, and we want to be able to use it either for the transfer of files between the two systems or for running CISL's MDS2400 as a remote printer through the PCO I/O daemon. A manual switch installed at CISL will determine whether the CISL end of the line is connected to the MDS2400's modem or to one in the IPC machine room; the other end will be connected to the Phoenix 6180.

This situation requires software that can detect the status of the switch and act accordingly. We also wish to be able to go back and forth between running the MDS2400 and the file transfer daemon as nearly automatically as possible. The proposed design will accomplish the transition as the result of a single command followed by manual resetting of the switch.

We envision a process on the PCO system which, by means of a special process overseer, will be prepared to run alternately as a remote printer (i.e. MDS2400) driver or as a file transfer driver. When this process logs in initially, it attempts to come up in normal fashion as a driver for the MDS2400, which it tries to attach; a "notify" mode is used in this attach call to indicate that if no recognizable response is received within a reasonable time (say 5 minutes), the DIM is to return with a non-zero status code. Thus, if the line is not connected to the MDS2400, or the MDS2400 is not powered on, the attachment will fail. The driver will then decide that it is to run as a file transfer driver, and will try to attach the line through the file transfer DIM, again using "notify" mode; if there is no response, the driver will again attempt to attach the MDS2400.

Unless the switch is set in an invalid position, or something is otherwise wrong at the CISL end, the Phoenix driver will eventually succeed in attaching something. Then, and only then, it will inform the coordinator of its existence and type; that is, the coordinator will regard the file transfer driver and the MDS2400 driver as separate processes, which just happen never to be running simultaneously.

Once the driver is running something, switching between states will be controlled from CISL, but the control will be exercised through the PCO driver process; this process will therefore be

the "master" process for the purpose of deciding who gets to transmit first. The mechanism for switching will be discussed later in this section.

Meanwhile, the file transfer driver at MIT (which could be logged in when the system came up) attempts in similar fashion to attach the line to PCO (without using "notify" mode); when appropriate messages have passed between the two systems, both drivers consider the line attached.

After the attachment is completed, the "master" driver asks its coordinator for a request. If there is a segment to send, it immediately enters "send" mode and proceeds as described in Section III; otherwise it sends a "nothing-to-send" message and enters "receive" mode. The "slave" driver, meanwhile, enters "receive" mode immediately on completion of the attachment. If both sets of queues are empty, the two drivers take turns checking their queues as described in Section III.

The switching mechanism for the line will work as described below.

To switch from running the MDS2401 to File Transfer

A new command, "switch", will be added to the remote driver command repertoire; it will instruct the driver to cease running the MDS2400 and become a file transfer driver. In particular, when a $SWITCH control card is placed in the MDS2400's card reader, the driver will complete its current request as usual, and then detach the MDS2400 and send a "logout" wakeup to the coordinator. In other words, from the coordinator's point of view, the MDS2400 driver will log out; in actual fact it will become a file transfer driver and attempt to reattach the line for file transfer. If this attachment succeeds, the file transfer driver will make itself known to the coordinator, and processing will begin as described above.

In order for the attachment to succeed, of course, the manual switch has to be in the MIT-to-PCO position; it will be the responsibility of the operator who inputs the "switch" command to reset the switch.

To switch from File Transfer to the MDS2400

A "switch_driver" command will be provided that sends a wakeup to the file transfer driver instructing it to cease file transmission and start running the MDS2400. Thus when the file transfer driver comes up, it creates an event channel whose name it places in a predetermined data base; in order to switch, some other process logged in to PCO issues the "switch_driver"

command, thereby sending a wakeup over this channel. (Access to the "switch_driver" command may be restricted in order to avoid accidents.) The file transfer driver will block on this event channel at the same time that it blocks for requests from the coordinator; thus it will be impossible to switch while a file is in the middle of being transmitted.

When it does receive the wakeup, the driver will send an "end-of-session" message to the file transfer driver at MIT and send an answering wakeup to the process that issued the "switch_driver" command, which will type a line advising that the manual switch should be reset. The driver then attaches the MDS2400 and comes up again as a remote printer driver.

MIT_drivers

Because there is only one 4800-baud modem on Service at MIT, we cannot run the MDS2400 from MIT at the same time as the file transfer facility. In fact, once the file transfer facility is functional, we will probably only want to run the MDS2400 from MIT when PCO is down. Therefore the MDS2400 driver at MIT might just as well be logged in manually on the rare occasions when it is needed, rather than sharing a process with the file transfer driver. (1)

When the MIT file transfer driver receives an "end-of-session" mesage from PCO, it detaches the line and, after waiting a reasonable time (so the switch can be reset), issues another attach call, without the "notify" mode described above. It will then simply wait blocked until the PCO driver comes up again for file transfer (in response to a "switch" command to the MDS2400 driver in Phoenix).

VI. BRIEF SUMMARY OF PROGRAMMING REQUIREMENTS

Necessary_changes_to_existing_software

The remote-device driver software will have to accept the "switch" command; some mechanism (a separate entry point?) will be required to inform a remote-device driver that it is capable of becoming a file transfer driver.

Some new parameters will have to be invented for io_daemon_parms in order to identify file transfer drivers; it may simply be sufficient to define the additional device class,

---

(1) If we get a second modem at MIT, we could run the MDS2400 driver with "notify" mode and use it to run the MDS2400 from MIT during file transfer.

since the coordinator doesn't care what device is being run by the driver, and the driver will use little if any of the presently-existing device-type-dependent driver code.

New software required

1) Procedures to run the file transfer driver, some of which may be cribbed from existing driver code.

2) A file transfer DIM, some of which may be cribbed from the g115 DIM.

3) The "transfer_file" command, which will probably submit requests through dprint_.

4) The "switch_driver" command.

VII. SUMMARY OF DATA MESSAGES AND ORDER CALLS

Data Messages Between File Transfer Processes

Message	Meaning
"ready_to_attach"	driver has issued "attach" call
"attach_accepted"	acknowledgement of "ready_to_attach" message
"seg_header"	header containing description of segment to be transmitted
text	actual contents of segment
"end_of_seg"	transmission of segment is complete
"nothing_to_send"	driver whose turn it is to send has empty queues
"end_of_session"	driver is logging out or changing to MDS2400 driver; line will be detached
"ack"	positive acknowledgement, message received
"nack"	negative acknowledgement, previous message not understood and should be retransmitted

Order Calls Used by File Transfer Driver

Order	Effect	Contents of info structur
"send_header"	Send "seg_header" message	Contents of segment head (input)
"receive_header"	Block for response from other end, should be either "seg_header", "nothing_to_send", or "end_of_session"	Type of response and contents of header if any (output)
"nothing_to_send"	Send "nothing_to_send" message	none
"end_of_session"	Send "end_of_session" message	none
"ascii"	Enter "ascii" mode, i.e. data is to be sent or received as 4 ascii characters per word	none
"not_ascii"	Enter "not_ascii" mode, i.e. data is to be sent or received in 6-bit bytes	none