MULTICS STAFF BULLETIN - 109
PROPOSED NEW I/O DAEMON DESIGN

To: Distribution

From: Robert S. Coren

Date: July 27, 1973

This document describes a proposed new design for the I/O daemon, to be completed by the fall of 1973. The document is divided into the following sections:

I. A brief description of the problems presented by the current implementation of the daemon

II. A summary of the ways in which the proposed design attempts to correct these problems

III. An overview of the new design

IV. A description of the Operations interface to the daemon under the new design

V. The "history" of a dprint request under the new design

VI. A more detailed discussion of many features of the new implementation, including the management of "device drivers", console input and output, and the handling of errors and abnormal conditions.

This document assumes some familiarity with the present implementation of the I/O daemon. It should be noted that this document overrides MSB 106 (Operation of Remote Peripheral Devices) in all cases where they disagree.

## PROBLEMS WITH THE OLD I/O DAEMON

The present Multics I/O daemon, with one independent process per printer, has a number of basic problems. The most serious of these are:

1) All I/O daemon processes are driven by the same generalized queues. At present this results in inconvenience, since a daemon which is only running a printer can receive punch requests, and vice versa; with the advent of remote printers, it will become an intolerable difficulty.

2) The lack of systematic communication between two or more I/O daemon processes requires removing each request from the queue before it is performed, to ensure that no request is performed more than once. This requires special precautions

to avoid losing requests which are in progress at the time of a system shutdown or crash.

3) As a result of problems 1) and 2) above, requests which cannot be processed when they are received (for example, punch requests received by a daemon which has not attached a punch) must be replaced in the queue by the daemon itself. This degrades priority scheduling and results in inaccurate accounting.

4) Although each daemon process can run both a printer and a punch, it cannot run them simultaneously; the printer is idle while the punch is running, and vice versa.

5) Once a request is completed -- or is believed by the daemon to be completed -- it is gone. If a printer, for example, has been having ribbon or paper problems for the past four requests, and no one (including the printer software) has noticed it, the only remedy is to resubmit the requests.

The new design attempts to solve all of the abovementioned problems. The central feature of this design is the concept of <u>one</u> central I/O daemon process, called the "I/O Coordinator" (or "I/O Daemon Coordinator"), and a lot of subordinate processes called "device drivers." Each driver will run one device, and will be fed requests one at a time by the coordinator. This design meets the above problems as follows:

1) Each "device class" is fed from a separate queue (or group of priority-ordered queues). A "device class" is considered to be a pair [device type, location]; e.g., "on-site printer" would be one device class, "on-site punch" would be another, "remote printer at CISL" would be a third, etc. There can be any number of devices in a given class; each device would have its own "driver" process, but they would all be fed from the same set of queues. The dprint command will place requests in one queue or another in accordance with a new "-device" ("-dv") control argument.

2) Requests are read from the queues by the coordinator only, so there is no duplication problem. A request is not removed from the queue until <u>after</u> it has been completed.

3) No attempt will be made to process a request unless there is a driver (and a device) waiting for it. Besides, since requests are not destroyed until they have been performed, there is never any necessity for the daemon to replace a request in the queue.

4) In general, no two devices will be run by the same driver, so there is no reason why two essentially independent devices cannot run (effectively) simultaneously. (The

special case of a remote printer-punch combination, which is not strictly speaking two independent devices, will be discussed later in this document.)

5) Each request, after completion and deletion from the queue, will be chained on to a special list for a fixed period of time (like half an hour). Such a request will be repeatable any time within the specified limit (but will *not* be automatically redone if the system crashes while it is in the list). Half an hour after the completion of each request, an alarm will go off, and the oldest request in the list will be deleted.

We have considered the possibility of operating the various devices from a single process, but this idea introduces several problems which are unlikely to be solved by the time that we will need the new daemon to run remote devices. In particular, such a design would entail either the development of a new asynchronous I/O interface or the introduction of subtasks within a process (or both). Implementation of both of these features is being considered for the future, and, once they are available, it will probably not be excessively difficult to adapt the design described herein to a single process.

## OVERVIEW OF THE NEW DAEMON DESIGN

The I/O coordinator will be driven by wakeups coming from two directions: from the various device drivers (indicating that they are ready for work) and from dprint_(announcing the addition of new user requests to the queues). The dprint_ wakeups are essentially ignored unless there is a device of the appropriate class waiting for work; to facilitate this procedure, the drivers will signal the coordinator over higher-priority channels than those used by dprint_. (1)

The coordinator process will have a process-group id of IO.SysDaemon.z and will be created in the same manner as the present I/O daemon (presumably, in most cases, as part of an exec_com executed at system-start-up time). *Driver* processes will be called Output.SysDaemon.z (2) and will be created by

(1) The overhead associated with an ignored event wakeup is very small. It is estimated that even on a very busy system, I/O daemon requests are unlikely to come in at a faster rate than one every 15 seconds, whereas the cpu time required to respond to a wakeup on an event wait channel is about 8 milliseconds.

(2) Drivers that run remote devices belonging to users may be

operator "login" commands. The operator will subsequently instruct the driver process either to attach an on-site device and begin work, or to wait until it receives a "dial" command over one of a specified group of lines. (When it receives the "dial" command, the driver will do further checking, such as ensuring that the device is of the correct type, and possibly requesting a password.) (1) Input to and output from these processes will be routed through the Message Coordinator; the I/O coordinator will have "source" id of "io" and each driver's source id will be the same as its device id.

Once a driver process exists, the running of the device will be largely automatic; operator intervention will only be required in special circumstances. Such operator commands as are required will generally be addressed to individual drivers; the most usual method for issuing such a command will be to type it on the daemon console (which might be any of several consoles) to be read by the driver when it is next ready (see below).

## OPERATIONS INTERFACE

(Note: This section does not include the special handling required for a remote printer/punch combination, which is described in MSB 106.)

The coordinator will normally be logged in automatically in the course of system start-up, but it can also be logged in manually from any console connected to the initializer through the message coordinator, in the same manner as the present I/O daemon, i.e. by typing:

```
login IO SysDaemon io
```

but this command will be accepted only if the coordinator is <u>not</u> already logged in.

A driver process is created by typing (again from the initializer console):

```
login Output SysDaemon DEV_ID
```

given different project id's for accounting purposes; the details of this mechanism are not fully worked out.

(1) Presumably some of the drivers will be brought up at system-start-up time as well; the instruction to start work or wait for "dial" will then be canned in system_start_up.ec or admin.ec.

which will create a process to run device DEV_ID, and give it a source name of DEV_ID. (The device class name will be implied by the device id -- e.g., the first four characters of the device id might identify the device class.) This process will signal the coordinator when it is ready to run (either immediately or when it receives a valid "dial" command from the specified device).

The coordinator will immediately start feeding requests from the queues for that device class to the driver, which will execute them on the specified device. If the operator wishes to type further commands to the driver, he should type:

```
r DEV_ID COMMAND_LINE
```

If the driver is idle, it will receive the command immediately: if it is processing a request, it will receive the command when the current request is finished.

The command thus input can be any one of the following:

```
detach
```

this command effectively detaches the device and disables the driver. It can be used when some temporary problem arises on the device. The driver is inhibited from receiving output requests, but its process is not destroyed.

```
attach
```

This is used to undo a previous "detach" command; it restores the device to service and enables the driver to receive requests from the coordinator. It has no effect on an already attached driver.

```
logout
```

This command is used to terminate the driver process.

```
restart n
```

This command causes all requests performed by this driver which are still available, starting with the one to which it assigned the identification number *n*, to be redone. All devices of the same class will be eligible to perform these restarted requests, and they will take priority over regularly-queued requests.

If it is necessary to interrupt a driver in the middle of a request (for example, if a print request is producing reams of garbage), a QUIT should be signalled by typing:

```
quit DEV_ID
```

Printing (or punching) will be suspended, and the driver will await a further command. This command may be any of those described above, in which case the driver will behave as if the "kill" command described below had been input, and then proceed to execute the command typed. In addition, any of the following commands may be typed (again in the format "^ DEV_ID COMMAND"):

kill

The current request is aborted, but is saved in the list of restartable requests. The driver proceeds to the next request (if any).

cancel

The current request is aborted, but is not saved. The driver proceeds to the next request (if any).

restart

If input without an argument, this command causes the current request to be restarted from the beginning (as if it had just been received from the coordinator).

## THE HISTORY OF A DPRINT REQUEST

A user enters a dprint request in the same manner as before, except that an additional optional control argument ("-device" or "-dv") may be included. (This argument actually specifies a device class.) The request will be placed in the message segment which represents the specified priority queue for the specified device class, (1) and a wakeup will be sent to the coordinator. The event message will tell the coordinator which device-class queue to inspect, and the event channel will identify the priority of the request (as it does now).

The coordinator will first of all ascertain if there is a driver of the specified class waiting for work. If there isn't any, it will simply leave the request in the queue for later processing and go blocked again; the request will not be processed until a driver asks for work.

---

(1) The "-device" control argument will, of course, have a default value, which will probably direct output to an on-site printer (or punch). The number of priority queues, instead of being fixed at three as at present, will probably be made an installation parameter.

If a driver of the right class is waiting, the coordinator will read a request from the specified queue. (1) At this point it will allocate a <u>descriptor</u> for the request, which includes the return arguments from the message_segment_$read call. (These arguments, in turn, include a pointer to the message itself.) Other information placed in the descriptor by the coordinator includes the device id of the driver to which it has been assigned, and the priority queue from which it was taken. The coordinator then sends the driver a wakeup over an event call channel; the event message is the word offset of the descriptor in a predetermined communication segment (which the driver initiated when it first came up).

The driver, when it receives the wakeup, will process the request in much the same way as the currently-existing I/O daemon, including access-checking and accounting. (It will <u>not</u> delete a file for which the "-delete" option was specified.) It will also fill in additional information in the request descriptor, including the request's sequential identification number and various status information (such as abnormal I/O status codes, indicators of whether the request was killed or cancelled, etc.). When it has finished printing (or punching) the request, it sends a wakeup to the coordinator, passing it the offset of the request descriptor in the event message.

When the coordinator receives this wakeup, it will record the clock time in the request descriptor, and thread the descriptor on to the end of a list of completed requests. (The coordinator keeps static pointers to the head and tail of this list.) It then sets an alarm to go off half an hour (or whatever other interval is chosen by the installation) of clock time later for removing the descriptor from the "completed" list. It is at this point that the request itself is deleted from the message segment. Now, if the driver has indicated that it is ready for more work, the coordinator will inspect the highest priority queue for that device class. If there is a request in that queue after the one which it had remembered as "last read", it reads this request, sets up a descriptor for it, and passes it to the driver; otherwise it repeats the process with the next lower priority queue, and so on, until either an unprocessed request is found or all queues for the device class have been inspected.

Our original request, meanwhile, is sitting in the "completed" list. If the driver that performed it receives the

---

(1) The request read will not necessarily be the one for which the most recent wakeup was sent. The coordinator keeps a record of the unique id of the last message processed in each queue and, when called upon to inspect the queue again, simply reads the <u>next</u> message (if the last-read message has been deleted, the first message in the queue is read).

operator command "restart *n*", where *n* is less than or equal to the identification number of our request, the coordinator will feed all drivers for that device class from the "completed" list rather than from the message segment queues, until all subsequent requests originally processed by that driver (including our friend) have been redone. (Descriptors marked as "restarted" are not rethreaded into the "completed" list, but are left in their original positions; nor is a new alarm set after a restarted request is finished.)

When the half-hour alarm (set at the original completion of our request) goes off, the coordinator will free the *first* descriptor on the "completed" list, along wiht with its associated message, unless that request is currently being redone. (Before the descriptor and message are freed, the "-delete" option, if specified, is honored.) Since there is a one-to-one correspondence between request completions and alarms, the "completed" list will not pile up indefinitely.

If when the driver originally wakes up the coordinator on completion of a request, it indicates that the request was cancelled (by an operator "cancel" command), the request descriptor is *not* threaded into the "completed" list; instead, after deleting the message from the queue, the coordinator frees the request and its descriptor immediately.

## FURTHER IMPLEMENTATION DETAILS

### 1. Initialization of drivers.

When a driver process is ready to run, a special initializing procedure is invoked. This procedure will be responsible for creating an event call channel over which the coordinator may signal the driver, as well as initiating all data segments that will be used in communicating with the coordinator (these segments contain such things as structures describing individual drivers, request messages, request descriptors, etc.). In addition, it will attach its device through a DIM whose name is kept in a static table of device-class- and device-dependent information. Once it has done these things, it will place its device id and the name of the event channel in a reserved area (which it will lock from other drivers) and send a wakeup to the coordinator along an event call channel provided for informing the coordinator that a new driver process has been created. It then goes blocked waiting for the coordinator to finish initializing the driver description.

The coordinator, on receiving the wakeup, allocates a structure for communication with the driver, in which it places the driver's process id, the name of the channel over which it expects to be woken up between requests, and the information that was provided by the driver. It sends a wakeup to the driver to give it the offset of this allocated structure; the driver then unlocks the area where it put its device id, etc.

2. Input to drivers

Once the driver process has been initialized, it should always be doing one of two things: 1) performing a print or punch request, or 2) waiting to be sent a request from the coordinator or a command from the operator, whichever happens first. It is not desirable for the coordinator to allocate a descriptor for a request until it knows there is a driver ready to process it, nor do we wish to send a request to a driver which has meanwhile started to process operator input. Therefore, when the driver goes blocked for operator input between requests, it must have some way of waking up again if there isn't any. Since the solution to this problem is rather tricky, it is presented in detail below.

(1) Before it starts processing coordinator requests for the first time, the driver establishes an event call channel over which to signal itself, and issues a wakeup over this channel. It then calls ios_$read, thereby effectively going blocked for input. (1) If there is any input, ios_$read will return to the driver procedure that called it; if not, the process will receive the event-call wakeup.

(2) Upon receiving this wakeup, the driver will send a wakeup to the coordinator indicating that it is ready for work. It will then execute a return (to ipc_), thus going blocked again.

(3) The coordinator will check to see if it has a request available of the device class associated with the driver. If it has one, it checks a bit in the driver's communications structure to see if the driver is still ready (since it might have received input in the meantime). If it is ready, the coordinator sets another bit indicating that it is about to send the driver a request; it then allocates a descriptor for the request and sends the driver a wakeup.

(4) If before receiving its own call wakeup the driver gets its input from ios_$read, it will turn its "ready" bit off and process the operator command. If the command was "restart n" it will send a wakeup along a special "restart" channel,

(1) In the case of some devices, such as the G115 (and hence the Mohawk 2400), the DIM must go blocked and wake itself up again at regular intervals, but the net effect is the same.

and then go blocked for coordinator requests; if it was "detach" it will detach the device and wait for further operator commands (presumably either "attach" or "logout"); if it was "logout" it will log out. When the driver has completed processing the command, it will issue another call to ios_$read and go blocked again.

(5) When woken up by the coordinator, the driver, after performing the output request, sends itself another call wakeup (as before the first call to ios_$read), and goes blocked again. When it receives this call (or operator input if any) it will send a wakeup to the coordinator to tell it it has completed a request (so the coordinator can thread the request descriptor onto the "saved" list and delete the request frmom the queue).

(6) In the special case where the driver receives operator input after the coordinator has already set the "request-pending" bit as described in paragraph (3) above, the driver will pretend that it received the input after completing the next request. It does this by setting an "input-pending" flag and then going blocked on an event-wait channel. After it performs the output request in response to the coordinator's call wakeup, if the "input-pending" bit is on, it sends itself a wakeup over this event channel (before sending itself the call wakeup as in paragraph (5)) and process the input as described in paragraph (4).

Since event wait channels will have priority over event call channels, the driver will always receive its input eventually. In the case of urgent input, the operator can send a QUIT to the driver as described above, in which case it will wait for operator input only.

Appendix A contains flowcharts which may clarify the algorithm described above.

3. Remote printer/punch combination

This is a special case, since from the coordinator's point of view two devices of different classes are involved, but in point of fact only one of them can run at a time. The procedures used by the driver for this device will accordingly appear to the coordinator to be two separate drivers, one for the printer and one for the punch; but these two drivers will happen to have the same process id, and the process in question will ensure that no attempt is made to run both at once (by never setting itself "ready" for print and punch requests simultaneously). In fact, the relevant driver will be "created" (as far as the coordinator is concerned) when output for that device is first requested. In other words, when a $*$START_PRINT command (1) is read in for the first time, the coordinator will be informed of the creation of a remote-printer driver; similarly, when the first $*$START_PUNCH command comes in, the coordinator will be told that a process is ready to run a remote punch. The $*$START_PRINT_PUNCH command will produce a "punch driver" if and only if there are no print requests pending, and conversely for $*$START_PUNCH_PRINT.

A flag in the driver structure will indicate whether or not the driver wants to be woken up after a completed request even if there are no requests pending; this will permit the remote printer/punch to be ready for the second device if the first one's queues are empty.

## CARD READING

There is no reason, actually, that the coordinator should ever have to know about card-reading. A "login" command for Input.SysDaemon would cause a process to be created which would attach the card reader and wait for card input. "Detach" and "logout" commands could be input to it in the same manner as for any other driver.

The card reader attached to a remote printer/punch is again a slightly special case. When it receives a $*$READ_CARDS command, the driver process will indicate that neither the printer nor the punch is ready, and accept input from the reader. The coordinator will hear nothing further from the device until output is again requested.

## MESSAGES FROM THE DAEMON

The drivers will have several different kinds of output to produce (aside from the printing and punching of user files): "normal operation" messages ("Request 75.2: Printing >doc>info>bugs.info for Gintell.Multics.a", etc.); warning or "information" messages ("Enter request:", "No punch requests in queue", etc.); and error messages (which category includes questions sent through command_query_, e.g. "Do you wish to cancel request 75?"). These three types of messages may want to go to three different places, and accordingly will be written on three different (or at least differently-named) streams. "Normal operation" messages will be written on "daemon_output" and will probably be directed to a file for later examination (if anyone

---

(1) The commands used to run the remote printer/punch are described in MSB 106.

is interested). Warning messages will be written on "user_output" and should probably appear on a terminal. Error messages will be written on "error_output" and <u>must</u> appear on a terminal. The routing of all these streams will be performed through the message coordinator; for devices in the machine room, "error_output" and possibly "user_output" for all drivers will presumably be directed to a single console, whereas for a remote printer/punch they will most likely appear on the printer (although a terminal may be dialed up if desired). [1]

The coordinator will probably produce error messages only: these will be written on "error_output" and may be sent to the same console as the on-site drivers' "error_output". In addition, certain serious error conditions will be recorded in the system log (see below).

## ERRORS AND ABNORMAL CONDITIONS

The primary goal in handling errors arising during daemon operation is to ensure that the requisite information is reported while disruption of normal operations is kept to a minimum. In any case where recovery is possible, the process which encountered the error will report and continue its work.

### 1. Errors while processing a request

These will be handled very much as in the present implementation. Missing or zero-length segments, insufficient access, etc. will result in the request being skipped; I/O errors and unclaimed signals during printing or punching (e.g. out_of_bounds because a segment was truncated or deleted out from under the daemon) will cause the request to be aborted. In either case, the driver will proceed to wake up the coordinator as if it had completed the request, but it will also place a status/error code in the request descriptor. The coordinator can use this code to decide whether or not to save the request in the "completed" list, and also in certain cases (e.g. "device not attached") whether the driver is still usable.

### 2. Space allocation problems

If there are a large number of very busy drivers, it is remotely possible that the area used for request messages will get filled up; in this case, an error massage will be written on "error_output" and in the system log. The coordinator will then

[1] Similarly, on-site device drivers will generally read "user_input" from the same console, and remote drivers will read from the remote card reader.

free the oldest request on the "completed" list (provided it is not in the process of being redone) and try the allocation again. If this error recurs frequently, it strongly suggests some systematic problem in the coordinator's space allocation procedures. Similar considerations apply if the space for request descriptors becomes full.

3. Message segment problems

Handling of bad or inconsistent message segments will be greatly facilitated by an entry point that is being added to the message_segment_ programs which will permit a sufficiently privileged process to find out if a message segment has been "salvaged". Whenever the first driver for any given device class is created, the coordinator will examine and reset the "salvaged" bit in each of that class's queues. If in reading a request, the coordinator gets a code of error_table_$badseg (indicating that the message segment was found to be in error and salvaged during that call), it will try to read the request again. If it gets the same code, it will complain loudly (sending BEL characters to "error_output", for example) and behave as if the desired message was not in the queue. Any time that a message segment is discovered to have been salvaged, a message indicating that requests may have been lost is written on "daemon_output" and in the system log. If retries are repeatedly unsuccessful, the best bet is probably to shut the daemon down and look at the message segment to try to find out why it is unusable.

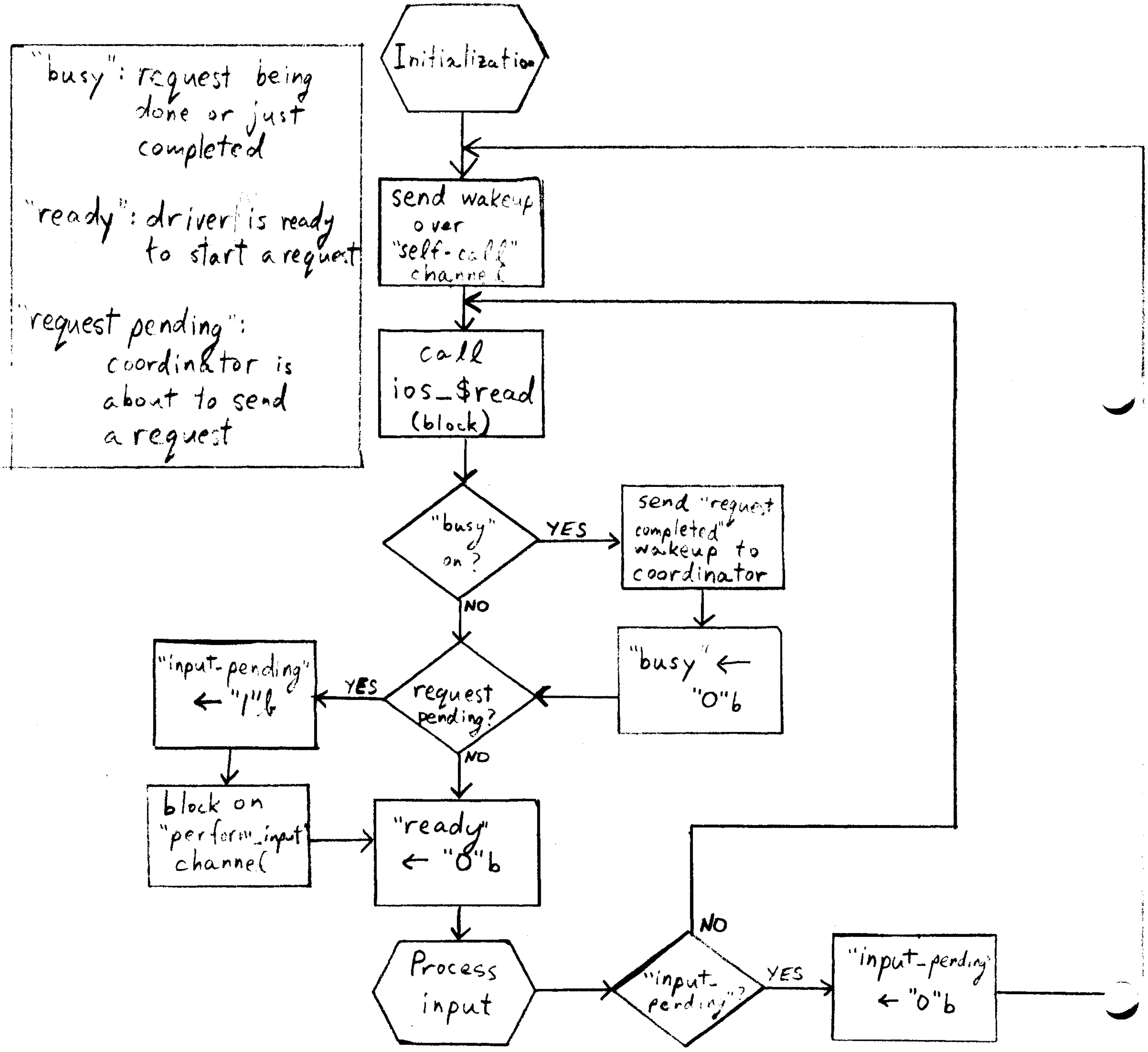
If a message that had previously been read is found to be missing when an attempt is made to delete it or read the next message after it, the coordinator will check to see if the message segment was salvaged. If it was not, it is probably safe to assume that someone removed the request (through the cancel_daemon_request command) while it was being performed.

4. Other errors in the coordinator

In the event of unexplained errors arising in the coordinator process (unclaimed signals, unrecognized event channel names, bad codes from file system primitives, etc.) a message will appear on "error_output" and the coordinator will go blocked waiting for the next event it is to service, after doing its best to ensure that its data bases are in a consistent state. Frequent occurrences of this kind probably indicate program errors in the coordinator itself.

# Appendix A

## Logic for Driver Input (part 1 of 3)

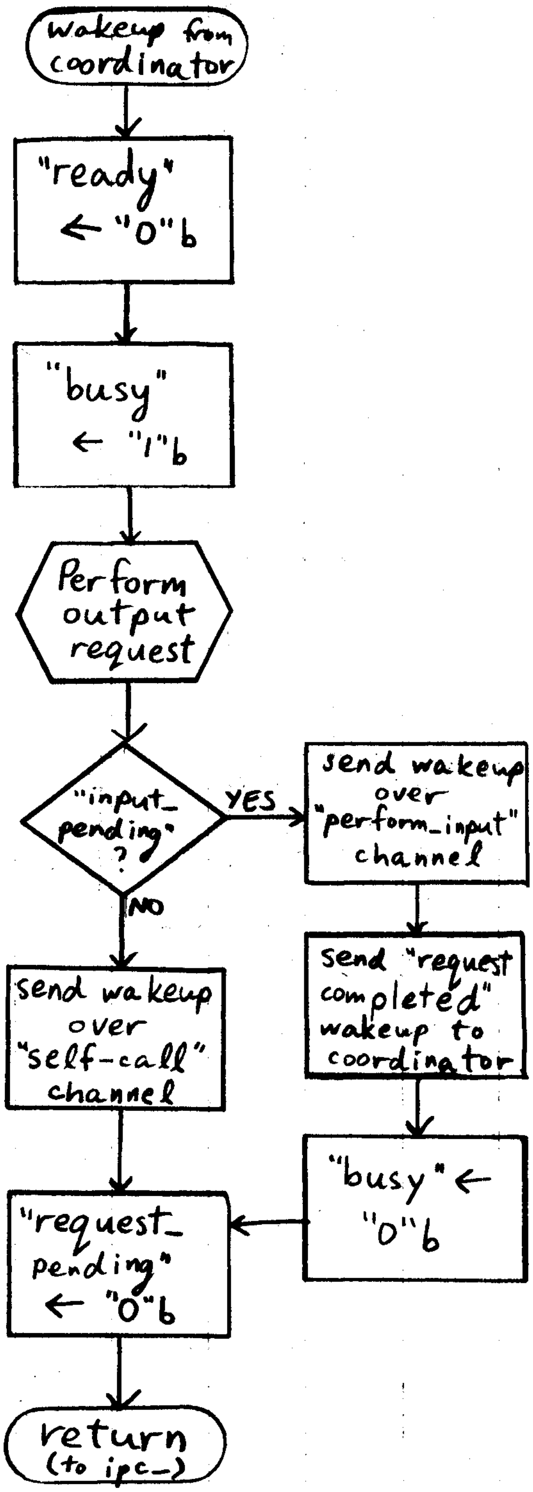


"busy": request being done or just completed

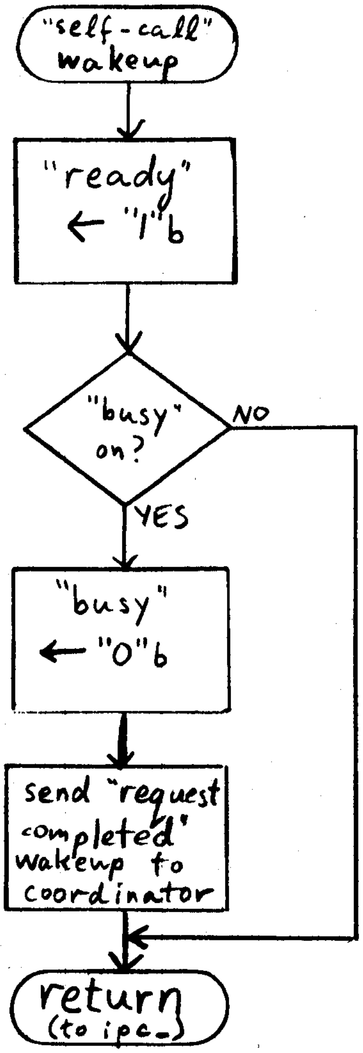
"ready": driver is ready to start a request

"request pending": coordinator is about to send a request

Appendix A
(part 2 of 3)



## Appendix A (part 3 of 3)


"self-call" wakeup
"ready" ← "1"b
"busy" on?
NO
YES
"busy" ← "0"b
send "request completed" wakeup to coordinator
return (to ipc_)