To: Distribution

From: Jerold C. Whitmore

Subject: Restructuring of the I/O Daemon Driver Software

Date: December 12, 1974

I. Motivation for the Restructuring

During our work in providing some new features for the Air Force, we needed to create a new type of device driver which would be controlled by the I/O Coordinator. However, due to the current structure of the driver software, we found this task very difficult. Specifically, it would be necessary to introduce a new operator response at IO Daemon initialization time and write new software to be executed for the remainder of the process. Limitations within existing modules, due to the assumptions about possible device types and how they would be controlled, prevent their use in new drivers.

It occurred to us that if we are having this trouble now, others will have the same problem in the future as more devices, which could be controlled by a device driver, are added to the system. For example, the MIT/IPC effort to write a spooling tape dim with a command interface shows one attempt to add a tape driver within the current structure. In the future we may want to add plotters, COM or other devices, which may be controlled by queued requests.

Therefore, now that we have a need for a new driver type, it will be easier to generalize the driver structure at this time rather than rewrite even more software in the future. This MTB addresses this generalization with the following goals:

1. Allow the easy addition of new driver software for new devices (or device classes) without recompiling existing software or changing the operator interface except for device specific functions.

2. Separate the driver control software into generic functional modules for easy maintenance and sharing or tailoring to new drivers.

3. Generalize the specifications of devices and device classes within io_daemon_parms allowing parameters to determine the control module which will be called.

II. Problems with the Current Driver Software

The statements made in the first paragraph were very general, so the following is presented as further justification. The reader may wish to look at the source programs in bound_iodd_.s.archive in the tools library to clarify some of the specifics.

1. All modules of the driver software assume either print or punch device specifics throughout. Extensive checking of new "switches" would be required if this software were to be used for new devices.

2. The format of a user request assumes printer or punch control functions which may not apply to other devices. (Only a small amount of data in the front of the structure is shared by the IO Coordinator.)

3. Current data areas in iodd_static assume that there will be no more than two logical devices per physical device (e.g., the printer and punch of a mohawk). This makes the addition of new multi-function devices more difficult.

4. Only the remote_ module knows how to find the two logical devices associated with a multi-function device using the "remote_tab" (which also limits the logical devices to "prt_name" and "pun_name"). This forces the single function and multi-function device initialization algorithms to be completely different. The limitation of multi-function devices to remote_ is also seen in io_daemon_parms.

5. The module input_cmd_, which does general driver command processing, assumes that only remote_ could have device specific operator commands.

6. To share code between the print and punch requests for local and remote drivers (and for historical reasons), the module output_request_ carefully formats printer head and tail sheets for punch requests and writes them through the discard_output_ dim.

7. The module iodd_listen_, which dispatches a user request (from the coordinator) to be processed by the driver, assumes that only one module knows how to handle the request, no matter what device is to be used.

In summary, the IO daemon driver is structured as an output only driver which knows how to print and punch with on site or remote (mohawk) devices. The dim_name in io_daemon_parms allows some flexibility, but not enough for completely new functions with the existing software.

III. Proposed Changes

At initialization time for any IO.SysDaemon the operator is asked whether the process is to be a "coordinator, driver, remote or cards." This will be changed to allow only "coordinator or driver", moving the distinction between remote and local drivers until later in the process so we can use some common general procedures for assignment of devices and device classes. ("cards" does not have to run as IO.SysDaemon. The removal of the "cards" option is the subject of another MTB.)

The current iodd_overseer_ will take on additional functions from io_daemon_driver_, remote_, and driver_init_, providing centralization of all initialization code and a uniform approach to searching the iod_device_tab. This is a necessary step since the "remote" response has been removed (above) and we must now determine the existence of multiple logical devices differently.

To accomplish this, we will introduce the concepts of major and minor devices to io_daemon_parms and to the iod_device_tab. A "major device" is a generic name associated with a physical device. A "minor device" is a generic name of a logical device associated with a major device. There may be up to ten (10) minor devices per major device. (Ten is an arbitrary number chosen to limit table size and still allow flexibility.)

Each major device corresponds to a physical piece of hardware attached to the process through a physical I/O or TTY channel. The per device attributes such as channel, dim, driver module, device dial id (for remote devices), and control terminal dial id (see MTB 129), are associated with the major device. Each minor device then has the device type and default device class associated with it.

Now, when the operator is asked to give the "device name and optional device class," he will specify the major device name (assume no device class for the present). The module iodd_overseer_ will then proceed by requesting the IO coordinator to associate each of the minor devices which belong to the major device with the driver process. Separate driver status segments will be created for each minor device. The driver may now behave as separate logical drivers for each minor device at its discretion. The driver may even ignore one or more of the minor devices. This does not tie up resources unnecessarily since they are all part of the same physical device and can only be attached to one process at a time.

There are two cases which can arise when the operator specifies the optional device class. First, when there is only one minor device for the major device (e.g., a printer connected to the IOM) the specified device class will override the default device class defined in io_daemon_parms. There is no ambiguity in the operator's intent in this case. The second case occurs when there are multiple minor devices for the major device. Currently, only the default device class for each device may be used. Since we do not know to which minor device the optional device class should apply, we propose that the operator be asked if the specified device class is to be associated with each minor device, in turn, giving the operator the ability to choose a different device class or retain the default for each minor device. This approach is chosen to simplify the operator interface for the common situation (in fact, it will not change at all) and still allow the operator to switch the processing of any device class to any driver which can handle it, local or remote.

A new data item will be added to io_daemon_parms, called "driver_module". This will be associated with the major device and will define the program to be called by iodd_overseer_ after the initial driver-coordinator protocols are completed. By convention, there will be standard entry point names in the driver module for initialization, request processing, command processing and condition handling. Entry variables will be added to iodd_static so each of the common driver subroutines (e.g., iodd_listen_ and input_cmd_) can have standard calls to perform driver specific functions.

New driver modules may need new data from io_daemon_parms. Therefore, to avoid modifying several programs and data structures for each new driver, the method of specifying the major and minor device attributes in the io_daemon_parms file needs to be generalized. Only those attributes which are needed during the common initialization of all drivers will have keywords in the parms file. Those attributes which may be more dynamic, on a per driver module basis, will be described by a quoted string after the new keyword "args". This allows the addition of new driver control modules to be completely parameterized...no recompilation should be necessary.

The format of the user request data structure which is stored in the message segment must also become more general. This will allow device options and driver options to be tailored to each driver module rather than changing one include file and recompiling all modules which use it (currently there are nine (9) external procedures which reference dprint_msg.incl.pl1). We will establish a standard header structure which will define that part of the request data which must be known by

the coordinator as well as all drivers. The "print_punch" variable will be changed to indicate the driver module which is expected to perform the request. Only the driver module and the command setting the value need to agree on the value (but it must be unique among driver modules). Each driver module will check to ensure that the request is meant for that particular driver. Then the remainder of the request data can be interpreted on a per driver module basis. The coordinator does not care how long the request is; it only needs the time, pathname, delete switch and version of the request header.

IV. Summary

These proposed changes provide the structure needed to completely generalize the driver software. A new type of device driver can be added by writing a driver module which meets the conventions and knows how to control the device. Then the io_daemon_parms file can be edited by an administrator to include the new device, device class and other attributes.

The operator interface will not change for the new driver except for new driver specific commands. Each driver can make use of the same overseer, which will handle all initial driver-coordinator protocol. Some of the driver subroutines can be directly used by the new driver module without changes; others can be easily tailored to meet new requirements.

These changes imply that almost every module of the current driver process will be either rewritten, restructured or removed. Driver modules for controlling the printer, punch and mohawk devices will have to be written. At installation time, all driver modules and io_daemon_parms will have to be replaced since they will now be incompatible with older versions.

APPENDIX I

IO DAEMON PARMS KEYWORD LIST - ORDERED BY OCCURENCE

/* PL/I style comments may appear anywhere in the parms file */

There are two keywords which specify global values for the IO coordinator and must appear at the beginning of the parms file:

Time: Defines the time in minutes that each completed request will be saved to allow for restarting. Normally, a delete option will not be completed until after this time has elapsed.

Max_queues: Defines the maximum number of message segment queues to be created or read for each queue group defined.

The next group of keywords are used to define the devices which driver processes may use. The device data is used by the IO coordinator to build the "device_table". All device definitions must appear ahead of the device class definitions.

Device: Defines the name of a major device (required - 32 char max)

driver_module: Defines the pathname or search name of the program which runs the device (required - 168 char max)

args: Defines an arbitrary character string for the driver module to decode which describes the major device. (optional - 128 char max)

channel: Defines the IOM channel of the device for direct attachment by the process. This must be specified if the dev_dial_id keyword is omitted and must be omitted if the dev_dial_id is specified. (8 char max)

dev_dial_id: Defines the dial_id to be used if the device is to be dialed to the process over a tty channel. Immediate attachment to a hard wired tty channel will be specified in the dial table if desired by the site. This keyword may not be specified if the channel keyword is used. (8 char max)

ctl_dial_id: Defines the dial id to be used for the control terminal to be dialed to the process. Optional - if omitted, no control terminal is required for the driver. (optional - 8 char max)

ctl_device: Defines the device name expected for the attachment of the control terminal. Example: ctl1 for the message coordinator, net103 for the arpa net, tty142 for the DN355. This keyword is primarily included for the message coordinator since there will be no check to ensure that this is the actual channel assigned. (optional - 8 char max)

minor_device: Defines the name of a minor device associated with the last named major device. If omitted, the minor device name is taken to be the same as that of the major device. (optional - 32 char max)

args: After the minor device keyword, args defines another arbitrary character string used by the driver module to describe the minor device. There may be one args keyword per minor device. If omitted for any minor device, a null character string is assumed. (optional - 128 char max)

default_class: Defines the default device class to be used for this minor device. If omitted, the operator must specify the device class during initialization. (optional - 32 char max)

like: This keyword is used to reduce the amount of text in the parms file when there are several major devices with similar attributes. All missing attributes in the specification of the major device, including minor device names, are taken from the major device name which is the value of the keyword. (optional - 32 char max) [Note: the major device named must have been previously specified if the file.]

The following keywords define the device classes used by the IO coordinator and drivers. The device class definitions must follow the definitions of the devices to help simplify the parsing of the parms file.

Device_class: Defines the name of a dynamic device class. (required - 32 char max)

device: Defines a minor device which may process requests of the last named Device_class. One or more instances of this keyword must be associated with a device class. The value is of the form major.minor to distinguish between minor devices of the same name in different major devices. If only one component is specified as the value, then major.major is assumed. (required - 65 char max)

driver_userid: Defines the only process group id which may be used to handle requests in this device class. If omitted, IO.SysDaemon is assumed. A personid of * is allowed, but the projectid must be specified. (optional - 32 char max)

accounting: Defines the pathname of the accounting procedure to be used for the driver. If omitted, or if the value of "system" is specified, the standard system accounting procedure will be used. The accounting procedure specified must be found during process initialization, or the driver will abort. (optional - 168 char max)

queue_group: Defines the message segment queues to be used for requests in this device class. If omitted, the queue group will be taken to be the same as the device class. Example: printer means use the printer_N.ms queues. (optional - 32 char max)

min_access_class: Defines the lowest access class request which a driver of this device class may process from the specified queue group. If omitted, the system_low access class will be assumed. The string must be acceptable to the convert_authorization_ subroutine. (optional)

max_access_class: Defines the highest access class request which a driver of this device class may process from the specified queue group. The access authorization of the driver must be equal or greater than this value. The max_access_class must be equal or greater than the min_access_class according to the rules of the access isolation mechanism. If

omitted, the value of min_access_class is assumed. The string must be acceptable to the convert_authorization_ subroutine. (optional)

min_banner: Defines the lowest access class to be used in marking the output generated by a driver process. If omitted, the value of min_access_class is assumed. The string must be acceptable to the convert_authorization_ subroutine. (optional)

End: This keyword has no value associated with it. It only serves to define the end of the parms file. (required)