

Device Driver Overview

Matt Genovese
Mark McDermott

Outline

■ Driver basics

- What are drivers?
- Where do they fit in embedded system design?
- What are the general responsibilities of drivers?
- What are application drivers versus OS drivers?
- How can the driver's client interface be architected?
- Discuss some basics of Linux drivers.

■ Details of Linux drivers

- Linux driver fundamentals
- Char drivers
- Memory allocation
- Char driver read and write
- Blocking
- Hardware access
- I/O ports
- I/O memory

Driver basics

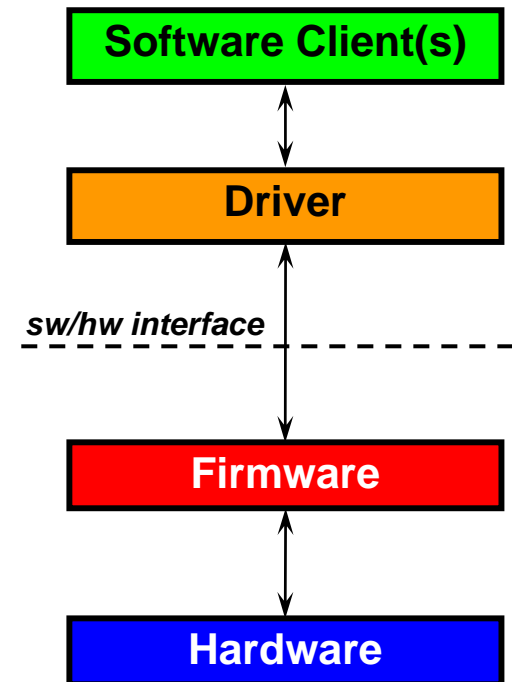
What is a driver?

■ Drivers are software...

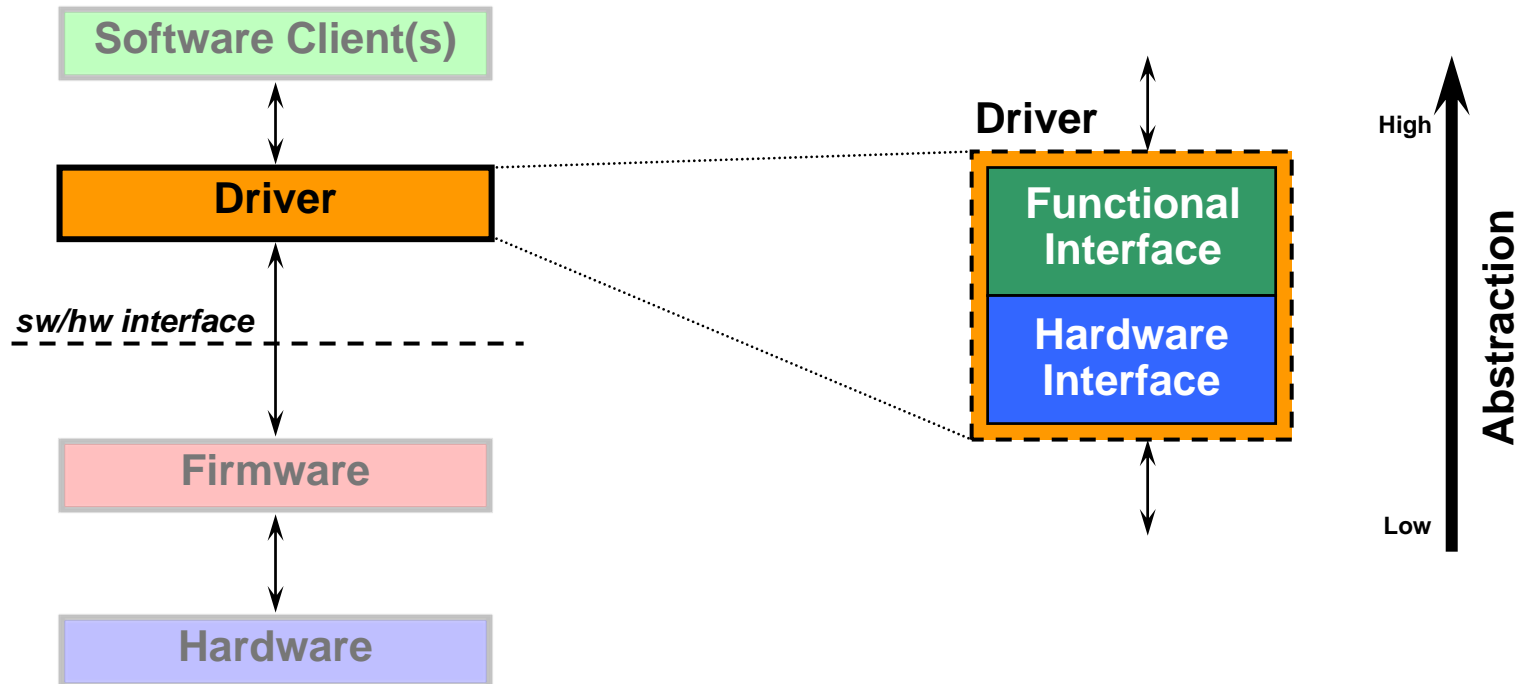
- Create a clean separation between upper-level client software and the hardware
 - Provide an abstraction layer above direct hardware communication
 - Manage bandwidth across the communication channel to the hardware
- Restrict the client interface state-space to only what is possible on the hardware
 - Control access to hardware from multiple clients
 - Ensure hardware errors translate into recoverable errors in the above software

■ Drivers are NOT applications.

- Applications are executable.
- Drivers are part of an application (library), or a module that registers itself with the kernel (Linux).



What is a driver?



- **Drivers conceptually have two interfaces**

- The *Functional Interface* interacts with the software application or OS kernel, receiving commands and returning responses.
- The *Hardware Interface* interacts with the hardware device to execute commands and receive results and status.

Drivers: Philosophy

■ Mechanism versus policy

- **Mechanism**: Make something at the lower level look like something else at the higher level. Emphasizes *capabilities*.
- **Policy**: Who can access it, what can be done with it, etc. Emphasizes *usage*.

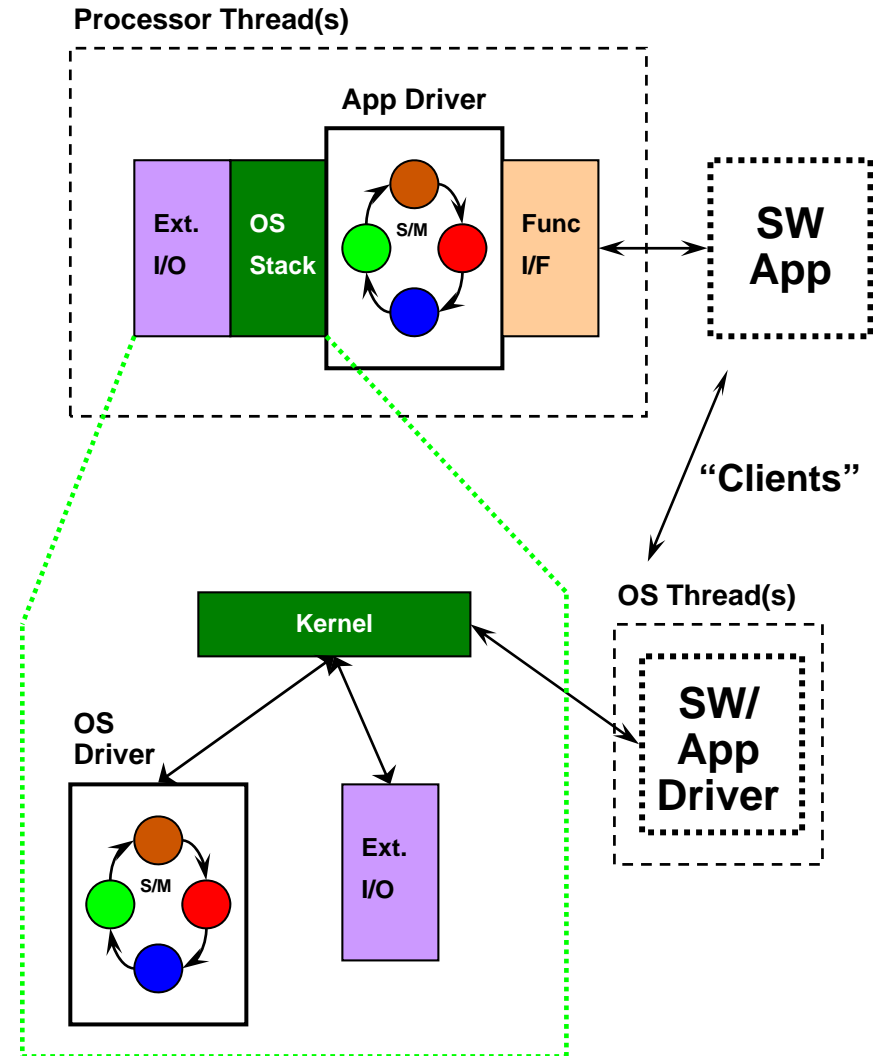
■ In general, drivers should implement the mechanism, and leave the policy to the OS or application.

- Drivers should implement the function-space of what is permissible and possible at the low-level.
- A driver that implements a policy also implements an inherent restriction on all possible applications above it.
- Very few instances when a driver should implement a policy.
 - When driver can affect global resources, multiple users, system stability, policy checks are present in the driver.

Drivers: Application versus OS

- **Application drivers cleanly separate and abstract device communication.**
 - Function at a higher level than OS drivers to further abstract communication based on an application's needs.

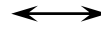
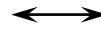
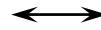
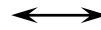
- **OS drivers enable safe and unified access to hardware.**
 - Accessed via kernel by software applications (e.g. application driver) to communicate with hardware.



Types of drivers

■ Application-level driver libraries

- The software application(s) are the direct clients of the driver.
- Applicable within a specific set of software applications.
- Represents a software architecture decision.
- Driver software interface and architecture is largely unrestricted.



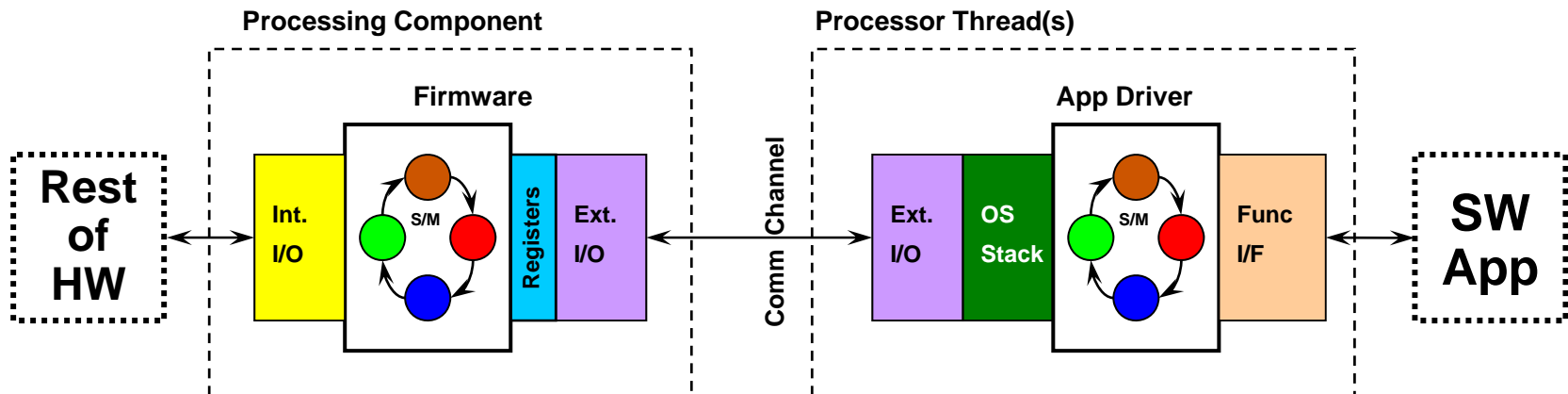
■ Operating system (OS) drivers (aka “stack”)

- The OS is the primary client of the driver, and applications access the driver via the kernel.
- Applicable within the OS, and any possible application written.
- Represents an OS stability and maintainability decision.
- Driver software interface and architecture is quite restricted.

Software communication with the hardware

- Firmware manages basic hardware and time-sensitive tasks, and provides a low-level software interface.
 - Small embedded memory footprint can limit functionality to providing the basics.
- Communication to firmware occurs over a communication channel (e.g. USB, RS232, SRIIO, SATA, etc.)

- Client software communicates with the driver via a high-level software interface.
- Application driver vets client requests and manages channel communication with firmware.
- The OS stack for the channel protocol is used for primitive communication with the hardware.



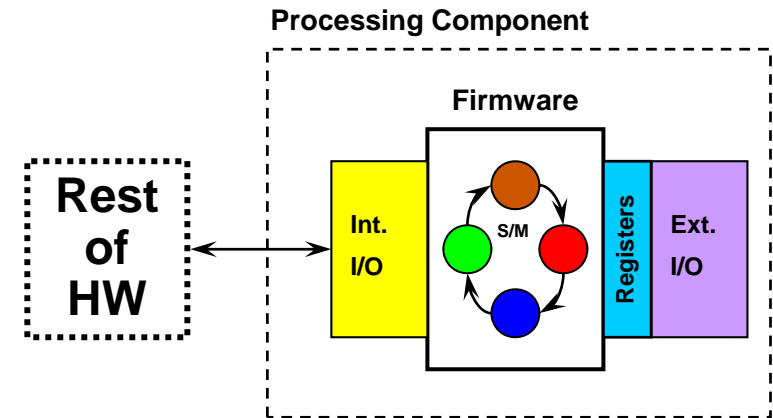
Firmware

Needs to manage...

- Reset
- Debug
- Basic functional operation
- Administrative tasks (e.g. upgrade)

Typically architected as a state machine

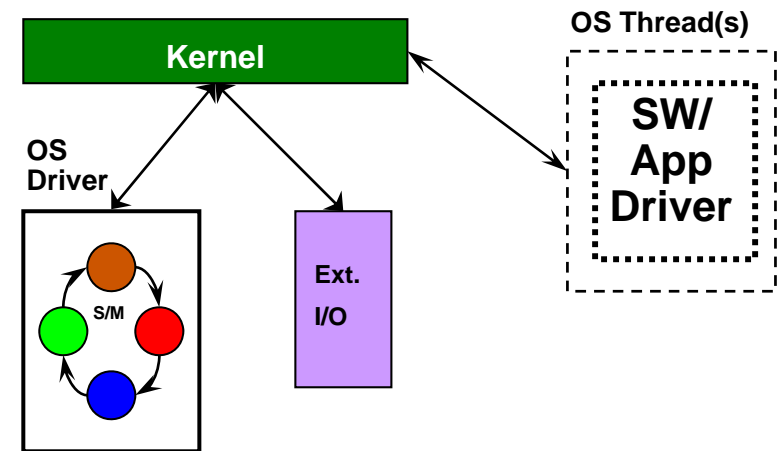
- Incoming commands from driver via the external I/O channel translate into state machine traversals, or message passing.
- Command status reported back to driver.
- Based on each device, firmware capabilities may be relatively basic and low-level, or could be more advanced.
- Error detection and basic error recovery capabilities may exist in firmware, but advanced recovery can be pushed up to the driver.
- If firmware is not employed on the hardware, then the driver must manage these responsibilities.



Linux OS drivers

Needs to manage...

- Reset, debug, functional operation, administrative tasks in hardware
- Error recovery
- OS driver specifications

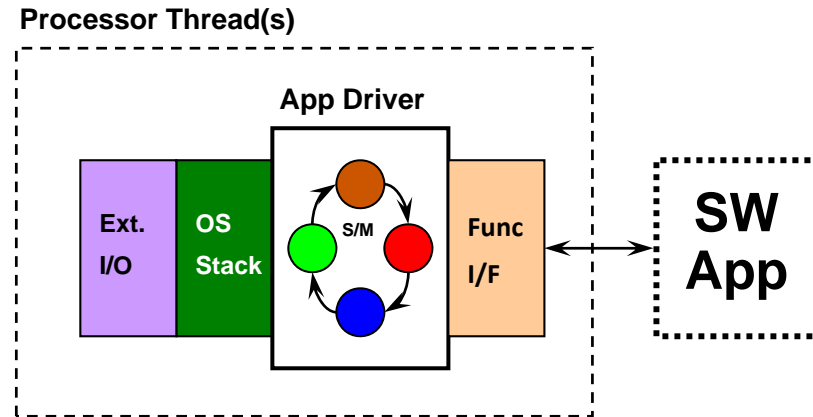


- Linux drivers are modules that communicate with the hardware in a structured, safe way.
- Provide implementation of base methods for communicating with the hardware.
 - Result: The device can be used by existing and new applications and other drivers seamlessly.
- Drivers may manage access to a communication channel or to a specific device.

Application drivers

Needs to manage...

- Reset, debug, functional operation, administrative tasks in firmware (not already managed by OS driver - device specific).
- Advanced debug and error recovery (for hardware, and client software)



Purpose: Provide an advanced feature set relevant to the specific device used by the client application.

- Functionality provided to application is higher-level than OS driver, combining appropriate low-level OS driver functions.

For application drivers, many architectures are possible.

- Dependent upon the needs of the intended client applications.

Client-driver communication: Commands

■ Definitions:

- A command holds the desired driver operation and inbound data.
- A response holds the command status/result and return data.

■ How does the client issue commands to the driver?

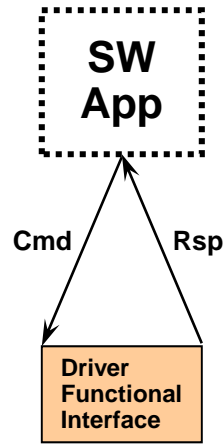
- Application driver: Likely C-style global functions to driver library
- Linux: C-style global functions to kernel

■ Application drivers: We have options for client interface:

- Command call: Client calls functions, each of which represents a command to the driver. Easiest for blocking drivers.
 - Pass in command as function's arguments, and the data + status are returned by that function.
- Command object: Use a single function for passing command objects. Useful for non-blocking drivers.
 - The command object is filled with the desired operation and data, and given to the driver.
 - After the driver executes the command, the response is encapsulated in or otherwise directly linked to that command object.
 - Advantage: Upgrade driver without changing command interface.

■ Linux OS drivers: Kernel registration; client calls system functions:

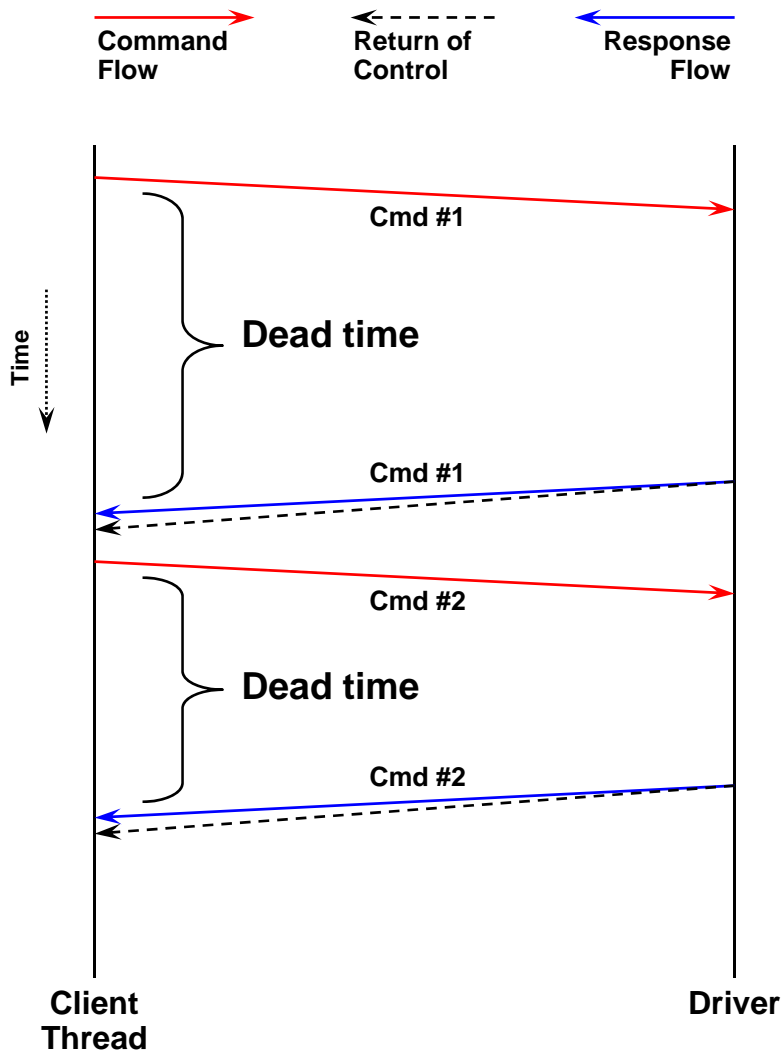
- More secure because kernel vets all access to drivers
- Enables categorization of drivers, and unified client interface to each
- Ensures privileged access (i.e. policy enforcement)
- This is the method Linux uses – discussed later.



Client-driver communication: Flow of control

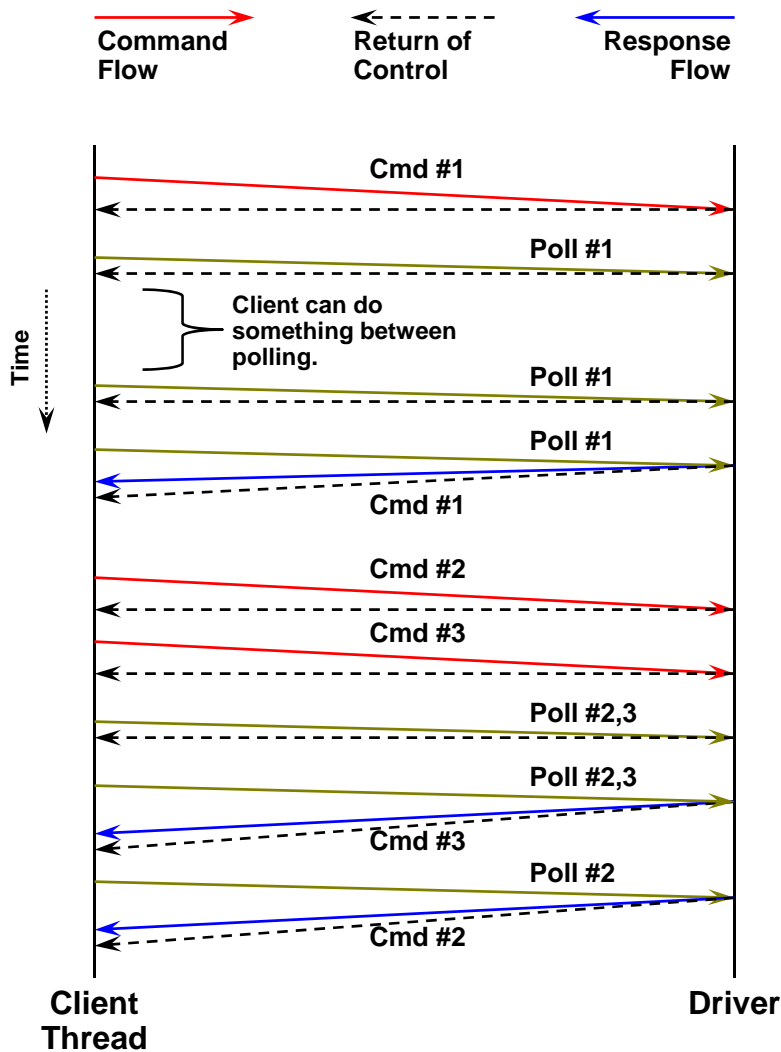
- **In general, there are a variety of ways a software client can interface with a driver in terms of program flow control. For example:**
 - **Blocking interface**
 - Call to driver blocks client thread execution until command is completed.
 - Easy to implement, but restrictive by forcing sequential operation.
 - **Non-blocking (NB) interface, client polling**
 - Call to driver returns immediately, even if command did not complete.
 - Client polls for command completion.
 - Allows for pipelining of commands, if supported by firmware.
 - Can create a blocking interface from a non-blocking interface.
 - **Non-blocking (NB) interface, driver call-back or software interrupt to client**
 - Same benefits as above (control returned immediately, pipelining).
 - Call-back: Requires client registration with the driver.
 - Driver alerts client when command has completed; no polling required.

Client-driver interface: Blocking



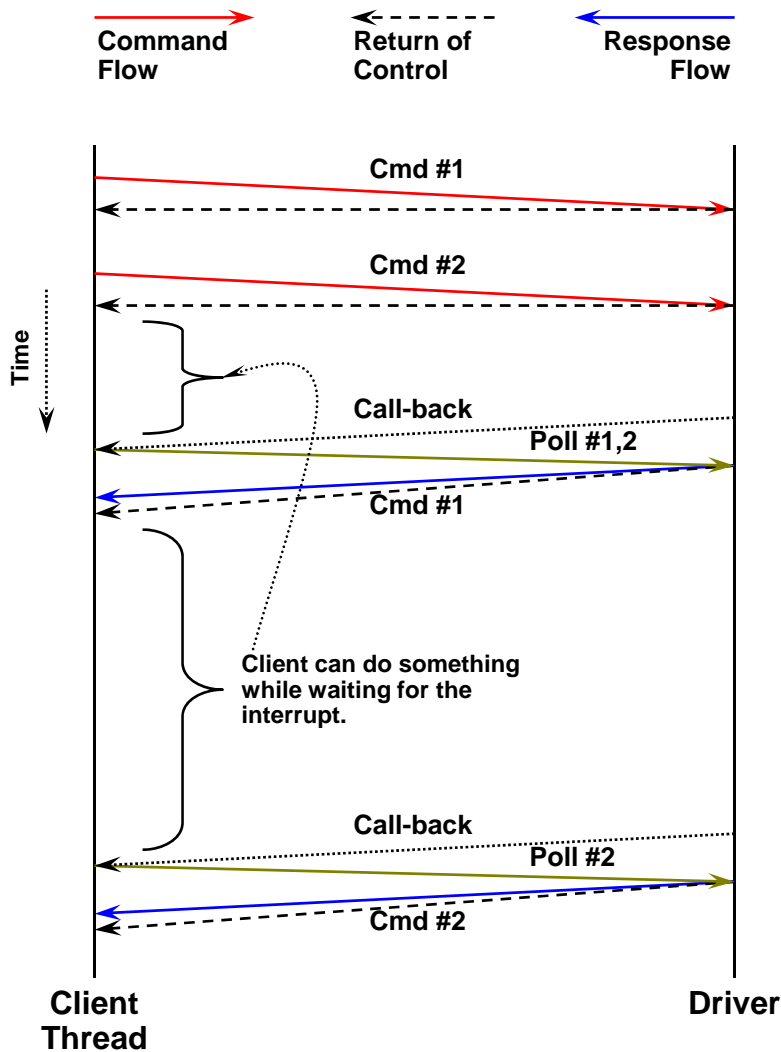
- Client commands to driver will block client thread until the driver fully executes the command.
- When the command is fully executed by the driver, the response is returned and the command has completed.
- Program control flow is coincident with command and response flow.

Client-driver interface: NB with polling



- Client sends command to driver, and receives control back immediately.
- Meanwhile driver executes command, and eventually queues up the result.
- Client polls driver to see if command completed.
 - Client has control: client can do other things while waiting.
- Client can pipeline commands, if driver allows.
 - Driver may enforce in-order execution, or all out-of-order for specific/all commands.

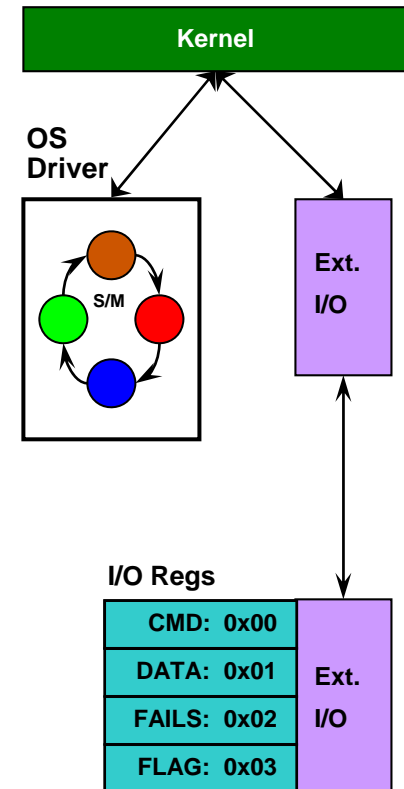
Client-driver interface: NB w/ interrupt or call-back



- Same benefits as the NB with polling scenario.
- If software interrupt, driver hits interrupt when command completes, and client polls only once.
- If call-back, after command completes, driver calls client directly. Then...
 - Client polls for result, or
 - Driver passed back results w/ call-back.
- Relieves client from polling after commands are issued.

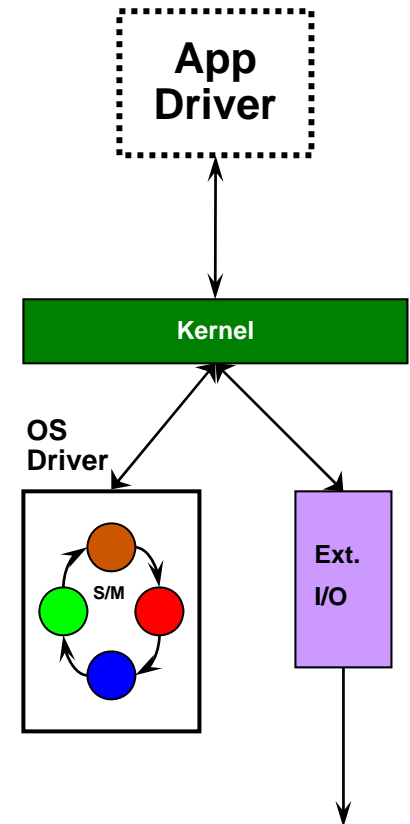
OS driver \Rightarrow direct hardware communication

- **The OS driver communicates with registers via the I/O channel provided by the OS.**
 - Devices may be memory mapped, or port accessed (depending upon processor architecture).
 - 8/16/32-bit hardware registers are accessed to configure and monitor state, and initiate activity.
 - Responsibility of OS driver is to translate basic commands into individual I/O register accesses that execute them on the hardware.
 - OS stability should never be compromised.



Application driver \Rightarrow OS driver communication

- **The OS driver has taken care of basic communication with the hardware.**
 - OS driver interface is basic and standardized for all Linux devices (of a given driver category)
- **Application driver now uses OS driver to...**
 - Build higher-level commands relevant to the upstream client (e.g. using command-response methodology).
 - Create program control interface(s) applicable to the upstream client (blocking, NB varieties).
 - Provide “nice” error recovery via timeouts, etc.
 - A crash of the hardware or firmware should not crash the client application.



Application driver \Rightarrow firmware communication

- For hardware that employs firmware, the application driver communicates with the firmware to execute commands.
 - Driver may communicate with the device's firmware via messages (higher-level than register accesses).
 - Device firmware may be written using a simple state machine approach. Driver takes firmware through states to execute commands.
 - Driver manages unique functionality of the device that is not directly managed or understood by the OS driver.
 - Prevent clients from “getting into trouble” by restricting scenarios that cause bad states in firmware.

